USENIX FIFTH COMPUTER GRAPHICS WORKSHOP AUTUMN 1989

U

US

USE

U NIX

USENIX

WORKSHOP PROCEEDINGS

FIFTH COMPUTER GRAPHICS WORKSHOP

November 16–17, 1989
Monterey, CA

For additional copies of these proceedings write

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is $18

Outside the U.S.A and Canada, please add
$10 per copy for postage (via air printed matter).

Program and Table of Contents

Fifth Computer Graphics Workshop

Monterey, CA
November 16-17, 1989

Friday, November 17

Program Chair:

Spencer W. Thomas
University of Michigan, EECS Dept.
1101 Beal Ave.
Ann Arbor MI 48109-2210

MICROFABRICATION ON THE MACINTOSH

Carlo H. Séquin

Computer Science Division
Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

ABSTRACT

A series of programs have been developed to model anisotropic etching of crystalline substances. The modeling proceeds in stages: from the geometry of the crystal lattice — to the removal probabilities of individual atoms — to the etching velocity of faces of certain key orientations — to a complete etch-rate polar diagram — to a list of potential planes in which new faces can form — to the new offset surface resulting from etching for a specified amount of time. Interactive programs on the Macintosh have been used to try out conceptual approaches and to find reasonable values for some a priori parameters.

1. INTRODUCTION

Recent years have seen dramatic progress in the construction of micro-electro-mechanical structures with technologies derived from integrated circuit fabrication. Many laboratories have fabricated prototypes of miniature levers, gears, and even functioning electrostatic motors (Fig. 1) of sizes comparable to the diameter of a human hair.[1]

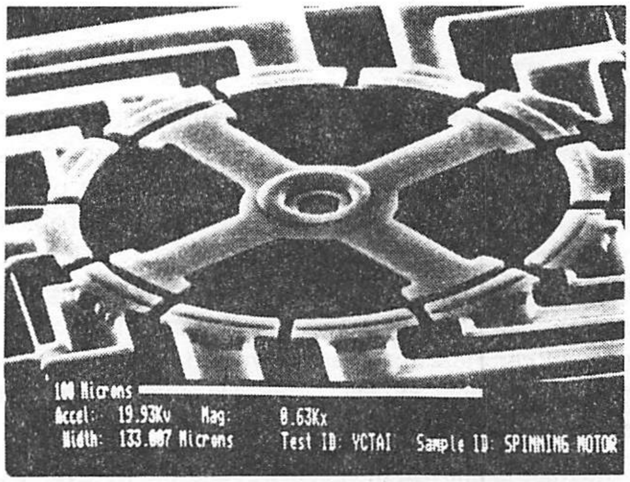

100 Microns
Accel: 19.93Kv Mag: 0.63Kx
Width: 133.007 Microns Test ID: YCTAI Sample ID: SPINNING MOTOR

Figure 1. *Electrostatic Micromotor, Courtesy of R. S. Muller, U.C.Berkeley*

The key techniques used in the fabrication of these structures are photolithography and selective and/or anisotropic etching. Shapes to be formed are covered with a patterned photo resist and the surrounding material is etched away. In this process preferential etches, which etch certain crystal directions faster than others, are often used to obtain almost perfectly flat surfaces or steep vertical walls on the desired features. Cantilevered features (Fig. 2) can be produced by underetching a persistent layer carrying the desired feature via some weaker, faster-etching layer.

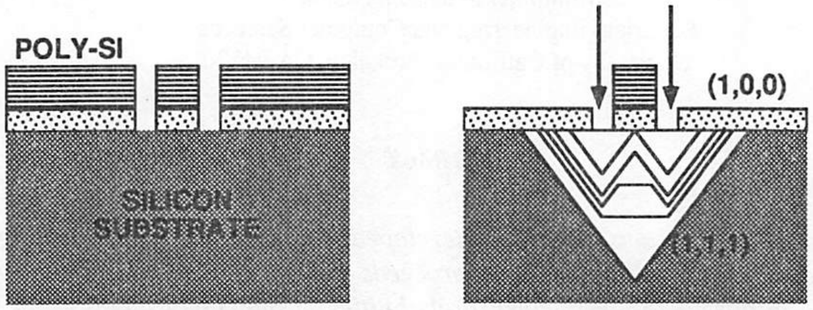


Figure 2. *Cantilevered Feature over Anisotropic Etch-Pit*

These processes are complicated, and the results are not always accurately predictable. Thus the successful design and implementation of such a micro-structures depends to a large degree on 'experience' or on the diligence used in a trial-and-error approach. As one contemplates to build more complicated micro-dynamical systems, such as gear assemblies or cutting mechanisms for micro-surgical applications, good predictive CAD tools become a necessity. At least two kinds of simulators will be required that permit one to determine suitable parameters for a system before it is actually fabricated. The first is a class of simulators to model the fabrication process itself, the deposition, growth, masking and doping procedures, and the selective and anisotropic etching steps. A second class of simulators will be required to model the kinematic and dynamical behavior of the resulting assemblies of geometrical parts. This paper is concerned with the first modeling step — in particular with the anisotropic etch processes needed to make sophisticated geometric structures.

2. ANISOTROPIC CRYSTAL ETCHING

Throughout this paper we will assume that we are dealing with homogeneous crystalline material with a constant orientation of its lattice. During etching, planar faces will remain planar and move parallel to themselves at some rate that depends only on the orientation of the face. Under those circumstances, it is also known that small surface elements of a given orientation and vertices between faces move through space on linear trajectories.[2,3] These etch-rates are conveniently described in a polar diagram in which the distance from the origin gives the etch-rate for every direction. Sometimes it is also convenient to plot the inverse of the etch-rate; this results in a "slowness diagram". For didactical reasons we present our approach to modeling the etching processes with a two-dimensional model — the concepts are easier to understand. In Section 10 we will then make the extension to three dimensions.

If the complete polar diagram is known, then constructing the new crystal shape after some specified etching time is just a matter of computing the offset surface with proper, orientation-dependent offsets. This conceptual statement glosses over some of the actual difficulties. The topology of the crystal can change: pits can etch through to another surface, holes can merge with other holes or cut into the outer contour. This needs some very careful management of the data structures describing the current crystal surface in order to maintain a consistent and correct representation. This task is made even harder by the fact that new faces may appear at corners because of differences in etch rates. With isotropic etching, the offset surface may also contain cylindrical or spherical parts. Thus even with full knowledge of all the etch-rates, it is quite challenging to make a robust program that will properly compute the new surface.

But etch-rates have been measured and published only for few materials in some select directions — or, at best, in all the directions that lie in two or three principal crystallographical planes.[4] An automated offset surface construction program requires complete etch-rate information for all possible directions. Unfortunately there is not enough information published to give us a rich set of different polar diagrams to test our etching modeler. Part of our effort has thus been directed at synthesizing reasonable and internally consistent etch-rate diagrams. We have developed a generator for plausible etch-rate diagrams based on geometrical considerations.

3. ATOMISTIC MODEL

To obtain physically plausible, consistent etch-rate diagrams, we use a simple atomistic model of the etching process. For this work, we assume that etching removes material only from the surface of the crystal and that the probability with which individual atoms are removed — and thus the resulting etching speed — depends on how easily the etching agent can reach and break the bonds between the atoms.

Experimental knowledge published in the literature reports etch-rates for various crystalline directions for various substances and etching agents. For certain anisotropic etchants such as EDP (a watery solution of ethylenediamine and pyrocatechol)[4] the dense planes of a silicon crystal (1,1,1) — having a face normal pointing in the (1,1,1) direction — are etched orders of magnitude more slowly than loose planes such as (1,1,0) or (3,3,1). We developed a geometric model, where tightly bonded, protected atoms inside a dense plane resist removal more strongly than loosely bonded atoms exposed at edges and corners.

To try out whether this model can lead to a reasonable macroscopic behavior, a simple simulator was written on the Macintosh. It permitted us to evaluate various ways of accounting for protection by geometric effects such as the density of the crystal plane and the protection offered near concave corners. This helped us build our intuitive understanding and to choose reasonable a priori values for the various bond strengths as a function of their positions with respect to the crystal surface (Section 5).

The simulator used a two-dimensional cellular array. We experimented with models that considered four, eight, or twelve nearest neighbors (Fig. 3a). The protectedness of an individual atom is computed from the number of it neighbors present. Nearest neighbors carry a much higher weight than second nearest neighbors, and neighbors in a dense plane also carry higher weights than neighbors in a looser plane. The sum of all these protection values is computed (Fig. 3b), and the probability that the particular atom gets removed is then a strong function of this neighborhood sum (Fig. 3c).

In each pass the simulator first checks each surface atom for its protectedness and computes a corresponding probability of being etched away. Once all surface atoms have been evaluated, they are removed with this calculated probability. Suitable color coding on the display screen makes it possible to inspect the calculated protectedness of the various atoms.

From these experiments it quickly became clear what the minimal level of sophistication would have to be to give reasonable results that agree with etching behaviors observed experimentally on real crystals.

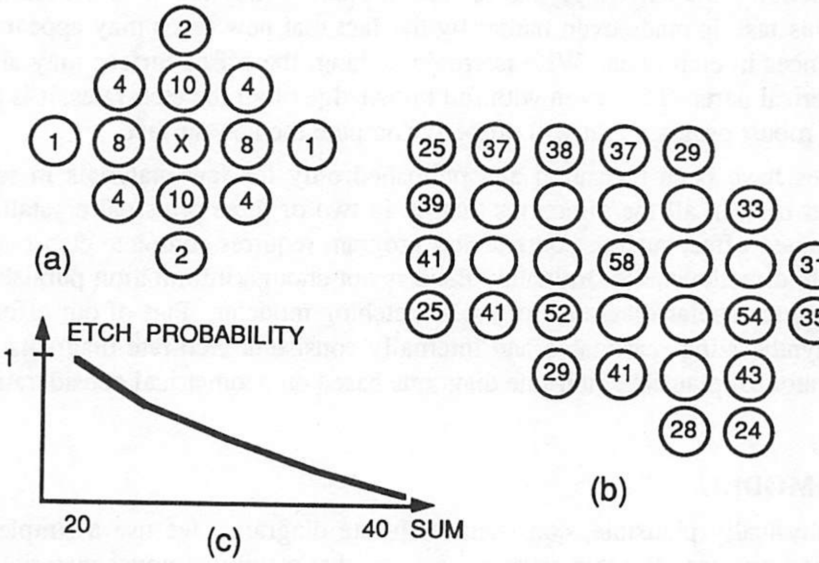


Figure 3. *Atom Removal Probability Calculated from Weighted Sum of Neighbors.*

It should be understood that many reasonable atomistic etching models can be conceived. With this work we do not attempt to shed any light on the electro-chemistry of the etching process; we just need a model good enough to give us plausible polar diagrams that can then be used to develop and test our geometrical offset surface generation programs.

4. OVERVIEW OVER THE MODELING SYSTEM

The modeling of the etching process proceeds in five stages. It starts with a crystal lattice cell at an atomistic level.

1) Estimate the probabilities that some specific atom at the crystal surface will get removed based on the accessibility to the etching agent of the bonds holding that atom in the crystal lattice. (We assume the etching agent to be larger than the lattice constant.)
2) Calculate the etch-rates of certain key crystal planes based on the cumulative probabilities of atom removal for the atoms that make up this plane.
3) Compute the complete 4π polar diagrams (i.e., 2π in 2D) of the etch-rate and of its inverse with suitable weighted averaging of the known etch-rates for the above key directions.
4) Analyze the polar diagram and compose a list of directions in which new faces might appear at corners based on the extrema and the curvature of the polar diagram.
5) Construct a new consistent surface corresponding to a specified etching duration by checking every corner for new faces that might appear and then advancing all faces by the amount corresponding to the etch-rate of that particular plane orientation.

The whole modeling system is thus a collection of five transformation programs that perform one well-defined step each. This modular approach makes it possible to enter the modeling pipeline at whatever stage one wants. If a complete etch-rate diagram is available from experimental data, the modeling process can be entered directly at step 4.

In the following five sections we discuss the above steps in detail. In some cases, simple Macintosh programs with suitable visual feedback were used. They first helped us to gain an intuitive understanding of the nature of a transformational step in order to come up with an appropriate computational model. Later, we used them again to verify that the transformational computations indeed produced the proper results.

5. FROM LATTICE GEOMETRY TO ATOM REMOVAL PROBABILITY

In the first stage of our deterministic modeling pipeline we took a more analytical approach to determining the atom removal probability than we did in the Macintosh simulation. We assume an etching agent that is larger than the average distance between neighboring atoms. It is then plausible that bonds at a planar surface etch more readily than bonds near concave corners, and much faster than bonds reaching away from the surface into the crystal (Fig. 4a). The probability that an atom gets removed is equated with the product of the probabilities that all its bonds to nearest neighbors get removed, thus atoms at convex corners get removed more rapidly than less exposed ones inside intact planes or near concave corners (Fig. 4b) (Black atoms have removal rate = 0). Again, this is a simple and plausible model that satisfies our needs, but it does not claim to reflect any deep insight into the actual etching process.

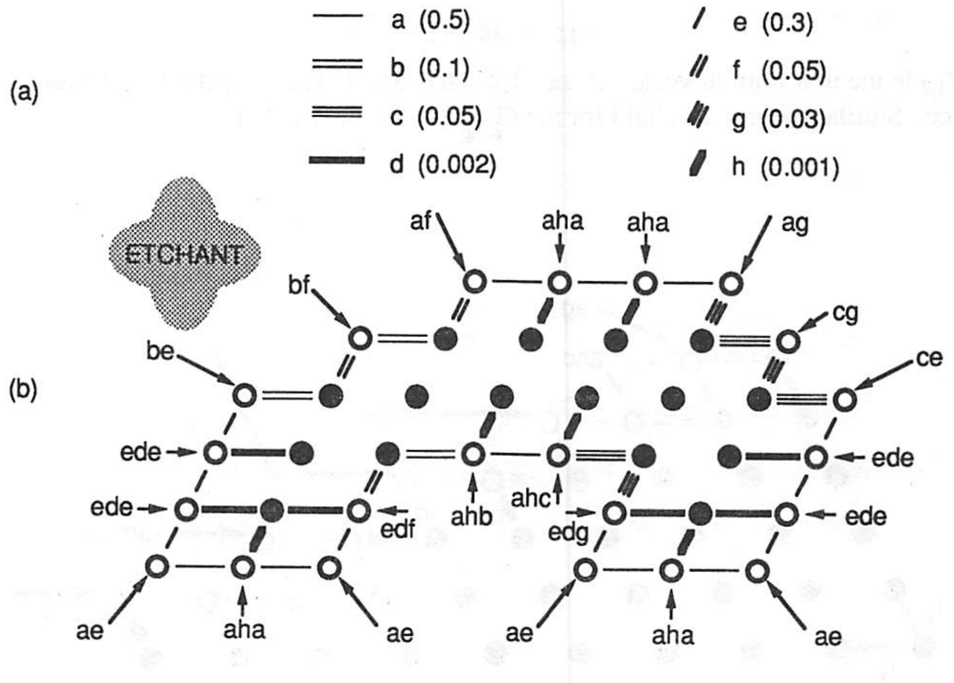


Figure 4. *Bond Breakage Probabilities (a) and Atom Removal Probabilities (b)*

For our experiments so far, the "bond breakage probabilities" have been assigned by hand. Later we plan to experiment with heuristic functions that derive such probabilities based on the size of the etching molecules and the geometrical obstructions in front of a particular bond.

Other models can readily be envisioned. The etching process could break bonds successively, and an atom would be etched away when all its bonds have been broken. It is not clear if such a model would provide a better approximation to reality. Because all five transformational modules have well-defined interfaces, any more sophisticated modeling of the actual etching process can later be substituted for these simple approximations.

6. FROM ATOM REMOVAL PROBABILITIES TO KEY ETCH-RATES

The subsequent macroscopic steps in our etching modeler rely on fewer assumptions. Once we understand the rates at which various atoms get removed based on their position at the surface of the crystal, we can calculate how a particular crystal plane is etched away. For an arbitrary plane this might be a complicated statistical computation. Thus we first concentrate on the basic crystallographical planes using simple geometric considerations to compute the etch-rates for these orientations. We will discuss this for two typical cases.

6.1. The (1,2) Plane

The (1,2) face of a 2-dimensional crystal (Fig. 5) is made up of atoms of two kinds: the corner atoms that etch away with very high probability (rate *ag* in Fig. 4), and the atoms in the middle of the "long" steps which etch very slowly (rate *ahc*) under our assumptions of an EDP-like etching agent. We can thus assume that all the corner atoms get removed more or less simultaneously, i.e., before any of the other atoms get removed. This means that in one "etching generation" the (1,2) face gets shifted in the $-x$-direction by an amount equal to one lattice constant $\vec{l_x}$, and this process requires $1/ag$ time units. This corresponds to a net etch-rate in the direction of the face normal $n_{1,2}$ of the (1,2) face of:

$$r_{1,2} = ag\ (\hat{n}_{1,2} \cdot \vec{l_x}),$$

where $\hat{n}_{1,2}$ is the unit normal vector of the (1,2) face, and $\vec{l_x}$ is one of the base vectors of the crystal lattice. Similar expressions hold for the (2,1), (-1,2), and (-2,1) faces.

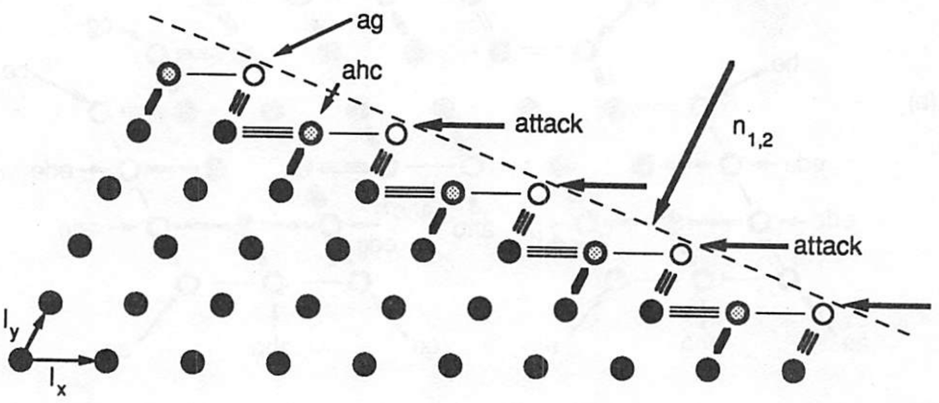


Figure 5. *The (1,2) Surface*

6.2. The (0,1) and (1,1) Planes

Even though the intact (1,0) plane contains only a single type of atom, the situation is more complicated here (Fig. 6). The removal of the first atom for such a surface is an unlikely event. But once the first atom has been removed, two corner atoms appear, and they have etch-rates substantially higher than the rate with which a first atom gets removed from the pristine plane. Thus, a layer of atoms will be stripped away fairly rapidly, exposing a new intact (1,0) face.

Since the chance of removing an atom from the intact plane is larger than zero, there will be a balance between the rate of attack on the perfect plane in all possible places and the rate at which the generated etch pits spread due to the lateral stripping action. The net etch rate for the (1,0) plane can be calculated to be:[5]

$$r_{0,1} = \sqrt{(af + ag)\, a^2h}\ (\hat{n}_{0,1} \cdot \vec{l_y}).$$

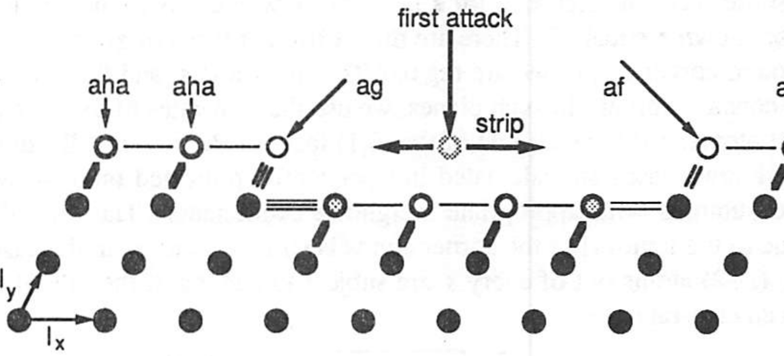


Figure 6. *The (0,1) Surface*

The computation of the etch-rate for the (1,1) plane follows the same scheme. It results in a net etch-rate of:[5]

$$r_{1,1} = \sqrt{(ec + ag)\, cg}\ (\hat{n}_{1,1} \cdot \vec{l_x}).$$

6.3. Discrete Etch-Rate Diagram

In a two-dimensional crystal, only the above three types of surfaces have to be calculated explicitly. Taking symmetry into account, one can readily determine the etch-rates for a total of 16 key plane orientations. A corresponding discrete etch-rate polar diagram is shown in Fig. 7(a).

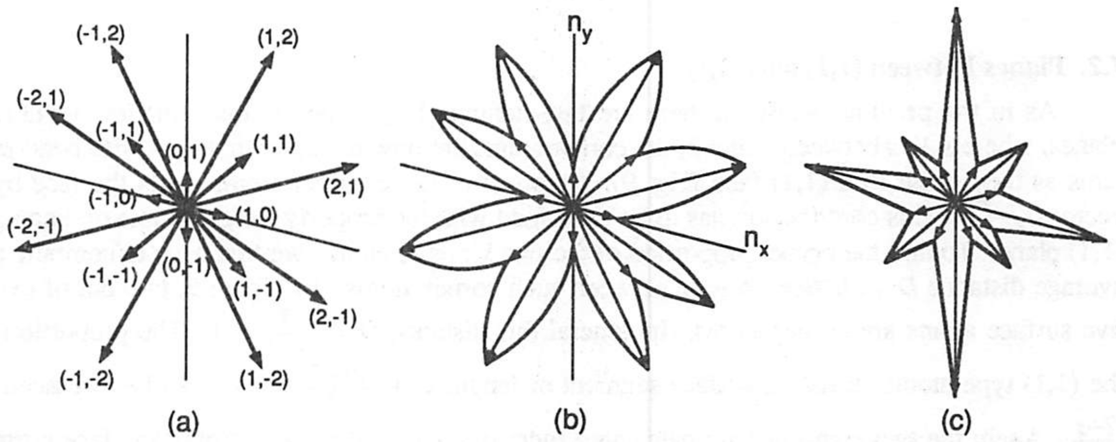


Figure 7. *Discrete (a) and Complete (b) Etch-Rate Polar Diagrams and Slowness Diagram (c)*

7. FROM KEY ETCH-RATES TO COMPLETE POLAR DIAGRAM

From the etch-rates in 16 key directions calculated in the previous section, the etch-rate for an arbitrary direction can be computed by using weighted averaging. This will be demonstrated on a couple of examples.

7.1. Planes Between (1,2) and (1,∞)

Figure 8 illustrates how the etch-rate for a face (±1,s), where s is in the range [2,∞), is calculated; for the case shown s equals 4. There are three different types of atoms to be considered. One is a loosely bound corner atom, two are regular (0,1) face atoms, and the last is a more protected atom near a concave corner. In such planes, we use the etch-rates of the pure (1,2) face for the first and the last atoms, and the etch-rate of the (0,1) face for the $s-2$ middle atoms. The two etch-rates for these known cases are calculated independently, projected onto the normal vector of the surface, and summed with appropriate weighting coefficients. The rate of lateral shift along the x-axis due to the removal of the corner atoms is still the same as in the case of the (1,2) plane. In addition, $(s-2)$ atoms out of every s are subject to etching at the rate of the (0,1) surface. This leads to an etch-rate of:[5]

$$r_{1,s} = ag\ (\hat{n}_{1,s} \cdot \vec{l_x}) + \frac{s-2}{s} \sqrt{(af + ag)\, a^2 h}\ \ (\hat{n}_{0,1} \cdot \vec{l_y})\, (\hat{n}_{0,1} \cdot \hat{n}_{1,s}).$$

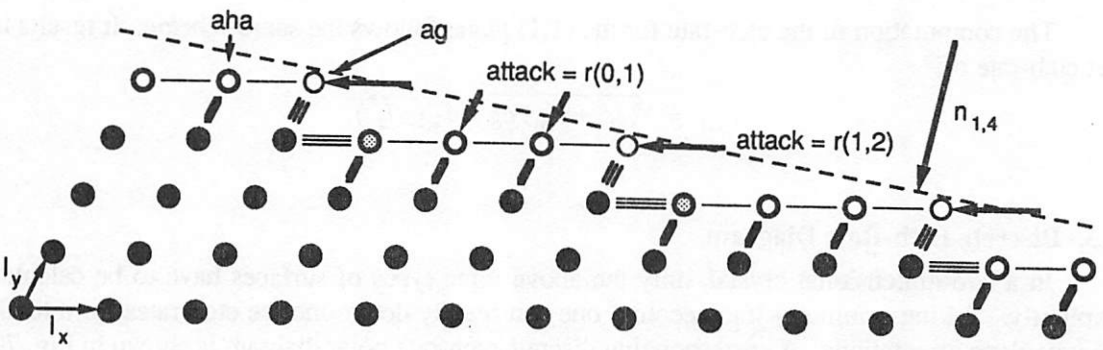


Figure 8. *Etch-rate Interpolation for (1,4) Plane*

7.2. Planes Between (1,1) and (1,2)

As in the previous analysis, there are fast-etching (1,2) corner atoms. But for the $(\pm 1,s)$ planes, where s lies between 1 and 2, the corner atoms are now mixed with atoms with bond patterns as they occur in a (1,1) face (Fig. 9). Removal of these corner atoms shifts the face by a vector $(\vec{l_x} - \vec{l_y})$. This contribution has to be combined with the properly weighted rate of etching a (1,1) plane. To find the proper proportion of the two kinds of atoms, we first have to compute the average distance D in lattice units $\vec{l_x}$ between such corner atoms. In Figure 9, two out of every five surface atoms are corner atoms. In general this distance D is $\frac{1}{s-1} + 1$. The proportion of the (1,1)-type atoms in such a surface segment of length D is $\frac{D-2}{D}$, which can be evaluated to $\frac{2-s}{s}$. Again the two etch-rates are calculated independently and projected onto the face normal direction, where they are summed with the above weights. This leads to a net etching rate of:[5]

$$r_{1,s} = ag\, (\hat{n}_{1,s} \cdot (\vec{l_x} - \vec{l_y})) + \frac{2-s}{s} \sqrt{(ec + ag)\, cg}\ \ (\hat{n}_{1,1} \cdot \vec{l_x})\, (\hat{n}_{1,1} \cdot \hat{n}_{1,s}).$$

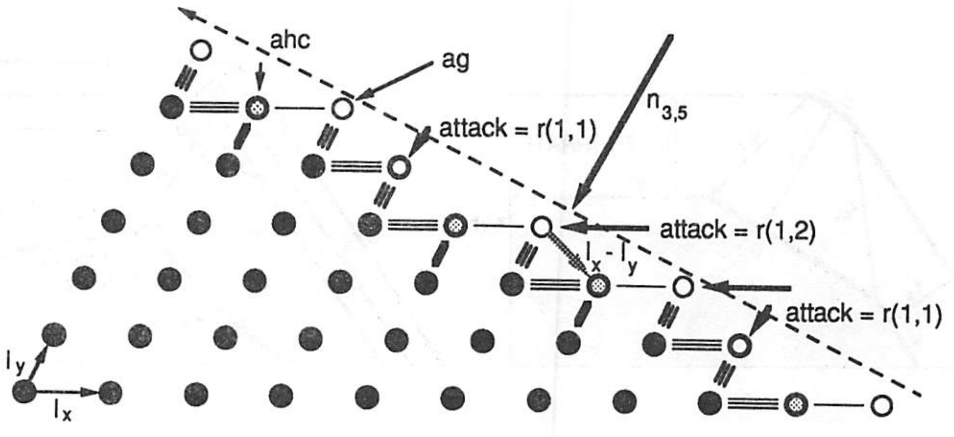


Figure 9. *Etch-rate Interpolation for (3,5) Plane*

7.3. Slowness Diagram

The results of the calculations carried out in the previous two subsections are now used to compose a complete etch-rate polar diagram. Enough points are calculated to produce a smooth curve as shown in Figure 7(b).

For the determination of an anisotropic offset surface, it is even more useful to plot the inverse of the etch-rate (Section 8), thus forming the slowness curve shown in Figure 7(c). In our example, the curve segments between the 16 key orientations turn out to be almost straight chords in the slowness diagram, whereas they are strongly curved in the etch-rate diagram.

8. FROM COMPLETE ETCH-RATE DIAGRAM TO VIRTUAL FACES LIST

8.1. Emerging New Faces

In anisotropic etching, the relative sizes of adjacent faces can change if the etch-rate differentials and the angles between adjacent faces have appropriate values. Such a case is illustrated in Figure 10(a) where face *c* is growing at the expense of its two neighbors. Such a face can even emerge out of a simple vertex between two faces *a*, *b*. This means that at convex or concave corners, new faces may appear that were not originally there. Under the right circumstances, e.g., isotropic etching, corners can also become rounded, which means that an infinity of new faces are introduced. We thus have to analyze the complete polar diagram to determine what new faces may have to be considered. Whether a face will actually appear at a specific corner, is then a question of what angles it forms with the adjacent two faces. This analysis, however, is done as part of the fifth transformation module.

Looking at Figure 10(b), we can derive the condition that a face is gaining on its neighbors to such an extent that it will actually grow in size. The analysis becomes particularly simple if the slowness curve is used. In general, a vertex *ab* located between faces *a* and *b* moves perpendicular to the chord between S_a and S_b in the slowness diagram;[2] in Figure 10(b) it would thus move in the direction S_{ab}. If there is a face orientation *c* between *a* and *b* that moves 'fast enough,' the slowness curve will have a concave corner between S_a and S_b. In this case, the two virtual vertices *ac* and *ab*, moving perpendicular to the two subchords, i.e., along S_{ac} and S_{cb}, will move apart as a function of time, and a new edge *c* will actually form between them.

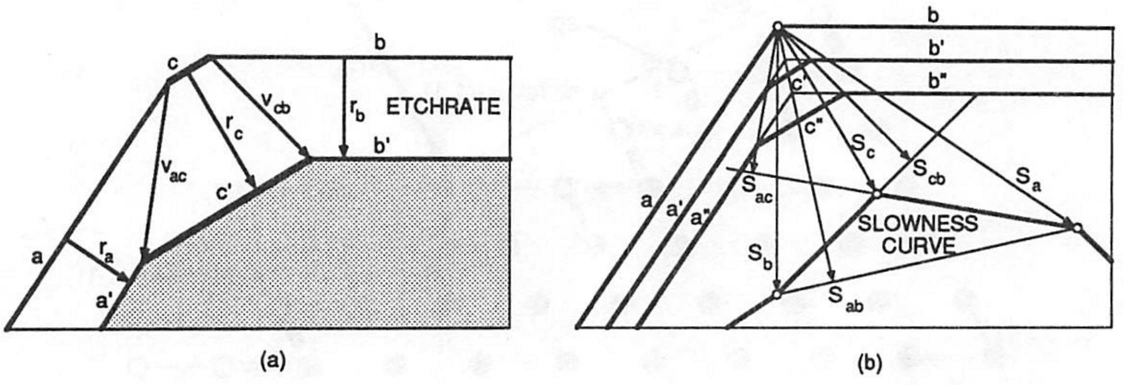


Figure 10. *Growing Faces (a) and Development of Virtual Faces (b)*

Similarly, it can be seen that for concave corners, the *slowest* etch directions will lead to convex corners in the slowness curve, which correspond to virtual vertices that move apart and which have the potential for forming new faces between them.

In our sample diagram (Fig. 7c), the eight segments adjacent to the four main crystal directions of type (0,1) are practically straight and will not generate any additional faces. The other eight segments adjoining the (1,1)-type orientations have a slightly convex curvature and will thus lead to new faces, i.e., some rounding, at concave corners. To keep the data structures and the computation time within reasonable bounds, we approximate the corresponding curves as a finite series of planar facets. A user-settable parameter chooses how many segments should be used for this approximation. Typically it is adequate to use about five segments between adjacent extreme points. Correspondingly, the etching of a corner corresponding to such a curve will result in a polygonal section with five segments.

8.2. Visualization on the Macintosh

The analysis presented above followed a more intuitive phase during which we experimented with some toy programs on the Macintosh. With the goal to understand under what circumstances new face would develop, we wrote a little display program — long before we had worked out the math using the slowness diagram. For every point drawn on an etch-rate polar diagram, the program would display a corresponding, suitably advanced face. The composition of all these faces then constituted the resulting corner geometry emerging from the originally given vertex; it showed readily what face orientations would make contributions to the new shape of the corner.

9. FROM VIRTUAL FACES LIST TO OFFSET SURFACE

The final step is to actually carry out the construction of the new crystal surface for some specified etching time. The construction of this offset contour starts with an analysis of each corner. The virtual faces list for all the angles between the two faces forming the corner are compiled, and the particular situation is analyzed to see which faces actually have a chance to grow; these are inserted into the data structure representing the current contour (surface). For each vertex in this contour, the direction of motion is determined. With these trajectories suitably parameterized in etching-time, the new offset surface can be constructed.

However, since the topology of the contour can change, we need to predict where vertex trajectories run into one another or into other faces. With this goal, we formulate the etching simulation as an event-driven process where the relevant events are the intersections of trajectories that may lead to topological changes. The expected etching-time at which these intersections would occur are calculated, and a corresponding event is placed onto the event queue. The topology update for the new offset surface is completed for the earliest etching time appearing in the event queue. Any such update may change the structure and may introduce new events into the queue. For any new corners created, the new faces list has to be analyzed and new faces may have to be introduced into the contour. This processing continues until there are no more topology-driven events on the queue with time stamps earlier than the etching termination time specified by the user.

10. THREE-DIMENSIONAL ETCHING

The same basic five transformations can be used to model the etching of three-dimensional objects. Conceptually they are not different from the steps described above for 2D crystals. However, the number of cases that must be realized is much larger, and the implementation becomes significantly more difficult.

10.1. Bond Strengths

For triclinic crystals the number of different bond strengths that would have to be considered increases from 8 to over 200. Thus we restrict ourselves to ortho-rhombic crystals, i.e., orthogonal lattices that may have three different lattice constants in the directions of the three axes. Even with that restriction, the number of different neighborhoods for bonds rises from 6 for the two-dimensional crystal to 81 in three dimensions.

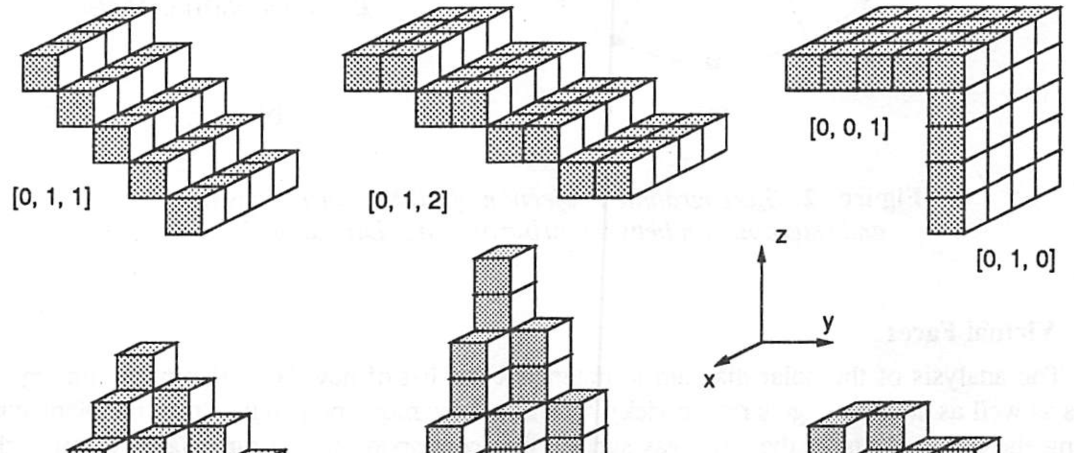


Figure 11. *Six Basic Crystallographic Planes*

10.2. Key Directions

Restricting ourselves again to ortho-rhombic crystal lattices, there are six key crystallographic planes that need to be analyzed. They lead to 98 key directions that can have as many as 19 different etch-rates (Fig 11). Again the Macintosh was a great help in visualizing the constellations of the atoms on these surfaces and in picking out the relevant bond combinations for the surface atoms considered.

10.3. Polar Diagrams

Interpolation to obtain the complete three-dimensional 4π etch-rate polar diagram is also more difficult. One now needs to perform a barycentric interpolation in the triangle between the three nearest key directions. In close analogy with the two-dimensional case, this can be done elegantly with the slowness diagram. Since the two-dimensional case has led to almost linear curve segments between the extrema in the 16 key directions, we thus fit 192 *planar* triangles between the 98 key direction points shown in the stereographic diagram in Figure 12(a).

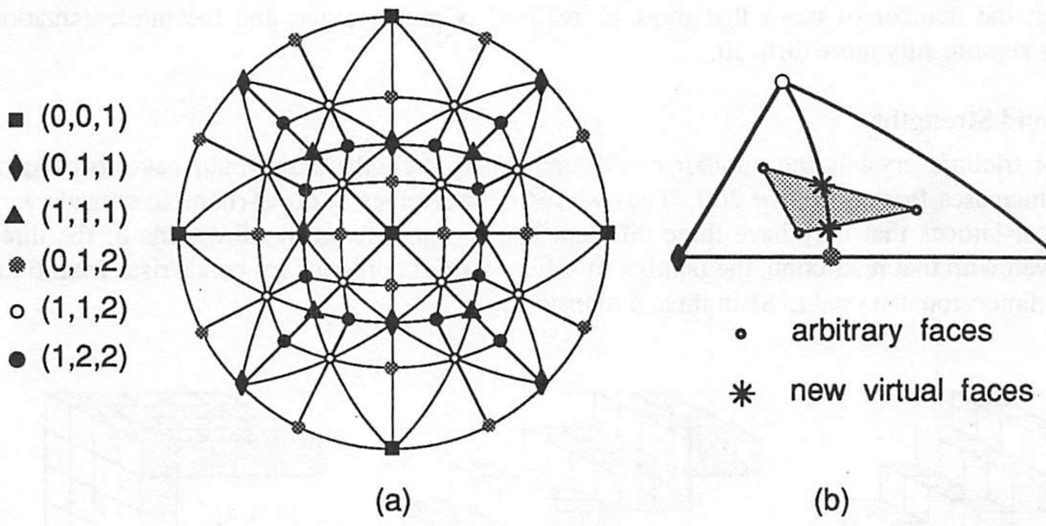


Figure 12. *Stereographic Projection of All Key Directions (a) and Interpolation between Arbitrary Face Directions (b)*

10.4. Virtual Faces

The analysis of the polar diagram to determine the list of new faces that might emerge at edges as well as at corners gets rather tricky. Vertices now move perpendicular to the plane connecting the three points in the slowness surface that correspond to the three faces forming the corner. For convex vertices, any 'inward bump' in the slowness surface within the triangle corresponding to the three given faces will produce a split of the vertex and corresponding growing new edges and/or faces. Since we assume the surface patches between the key points to be planar, the search for bumps is quite simple. The triangle (Fig. 12b) — or, in general, the polygon — spanned by the specific faces forming a given polyhedral corner is tessellated into planar facets along the edges shown in Figure 12(a). Each such n-sided facet then corresponds to a virtual n-sided polyhedral vertex. The trajectories of these vertices have to be analyzed to see whether any set of them move in a way that spans a new growing edge or face.

10.5. Offset Surface

In order to compute subsequent offset surfaces, the vertex trajectories are parameterize with the etching time, and the next 'interesting' event is determined, i.e., the time points when these trajectories intersect with each other or with other surfaces. In three dimensions, this computation becomes rather involved. Finally, the updating of the description of the advancing offset surface becomes an exercise in discipline. The special coincidences of geometrical features that could lead to program crashes if not properly anticipated are significantly richer in 3D than in 2D.

11. RESULTS

Figure 13(a) shows a result of etching a rectangular shape of a two-dimensional crystal. Figure 13(b) shows the underlying slowness curve; the eight (1,2)-type faces etch fastest and will dominate a convex shape after a certain etch-time. Superposed on Figure 13(a) are the paths taken by the various vertices showing clearly the splitting and merging of vertices and the corresponding appearance and disappearance of faces.

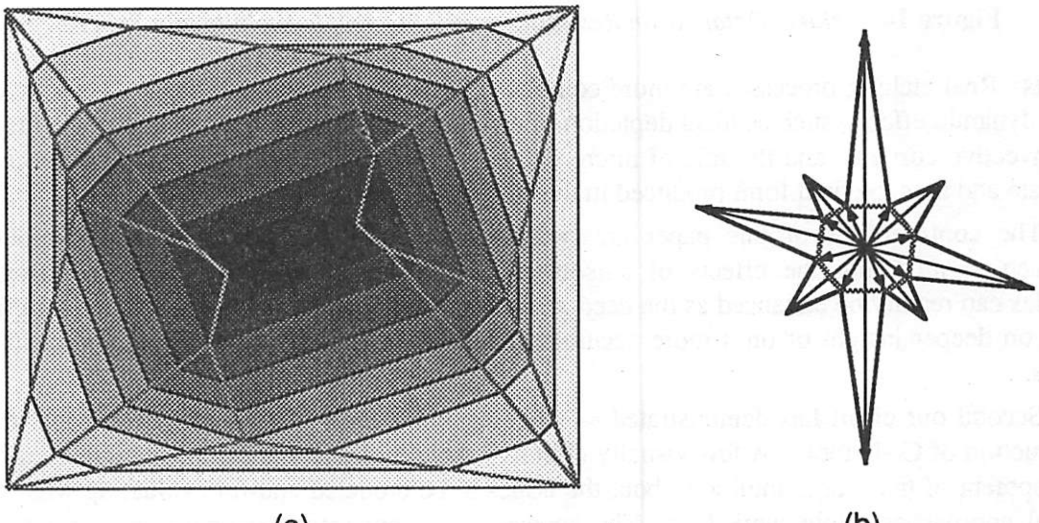


Figure 13. *Nested Etching Contours in 2D (a) and Corresponding Slowness Diagram (b)*

Figure 14 shows an example of a three-dimensional simulation. We assume that we subject a sphere of a crystalline material with a cubic lattice to an anisotropic etch with an effect similar to that of EDP on silicon. According to our simplified model, and in close analogy with the above two-dimensional example, the planes of type (1,2,2) will etch the fastest. Thus after a long enough etching duration these faces will not only dominate, but will have swallowed up all other faces. If we started from an original sphere of such crystalline material, the result would be the 24-sided Triakis Octahedron shown in Figure 14. It consists only of type (1,2,2) faces, and can easily be visualized as an octahedron with shallow three-sided pyramids on all eight faces.

12. CONCLUSIONS

The program sequence described here is only a modest beginning on the road to process simulators that can help the designers of micro-electro-mechanical structures gain confidence that their machines will work the first time, as is normally the case now with digital integrated

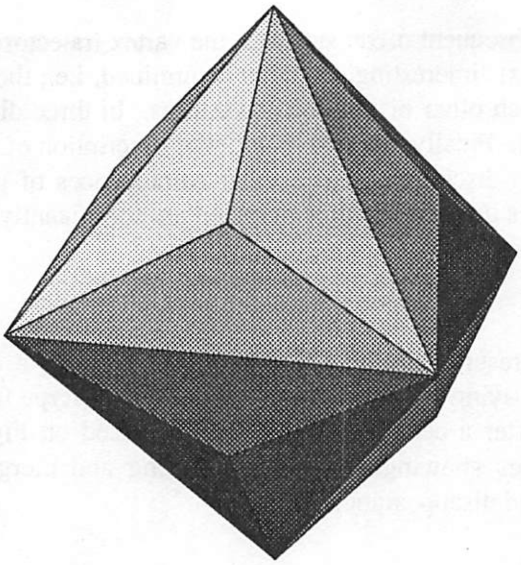


Figure 14. *Triakis Octahedron Resulting from Anisotropic Etching of a Sphere*

circuits. Real etching processes are more complicated than implied in this paper. They involve many dynamic effects, such as local depletion of etchant, the effects of temperature gradients and of convective currents, and the role of micro defects in the crystal — all of which influence the etch-rate and thus the final form produced in the etching process.

The contributions of this paper are two-fold. First we have demonstrated a modular approach to modeling the effects of anisotropic etching of crystalline material. Individual modules can readily be enhanced as the need arises or replaced with more sophisticated programs based on deeper insight or on a more accurate model of the underlying physical and chemical effects.

Second our effort has demonstrated — once again — the value for early feedback in the construction of CAD tools. A few visually effective skeleton programs can be invaluable for the development of the proper intuition about the issues to be modeled and for evaluating what conceptual approaches might work best. The appearance of powerful personal workstations with user-friendly programming systems that conveniently include color graphics, make the creation of such tools a pleasure.

13. ACKNOWLEDGMENTS

Many thanks go to Bill Foote who worked with me through the mathematics of these various programs. Readers who want more detailed information should ask for his forthcoming Masters report describing the development of these modeling program modules in more detail.[5]

Thanks also to Eric Enderton and Henry Moreton for reading this paper on short notice and for providing valuable comments.

Finally I would like to thank Tektronix for their continuing support of our modeling efforts.

14. REFERENCES

[1] R. S. Muller, ''Microdynamic Structures on Silicon,'' Proceedings of VLSI'89, Munich, F.R.G., pp 487-493, (1989).

[2] F. C. Frank, ''On the Kinematic Theory of Crystal Growth and Dissolution Processes,'' in ''Growth and Perfection of Crystals,'' Wiley and Sons, New York, pp 411-419, (1958).

[3] D. J. Barber, F. C. Frank, M. Moss, J. W. Steeds, I. S. T. Tsong, ''Prediction of Ion-bombarded Surface Topographies Using Frank's Kinematic Theory of Crystal Dissolution,'' J. of Materials Science, 8, pp 1030-1040, (1973).

[4] H. Seidel and L. Csepregi, ''Studies on the Anisotropy and Selectivity of Etchants Used for the Fabrication of Stress-free Structures,'' 1982 Spring Meeting, Electrochemical Society, Montreal, Canada, Abstract No. 123, (1982).

[5] W. F. Foote, ''Modeling of Anisotropic Crystal Etching,'' Masters Report, U.C. Berkeley, in preparation (1989).

3D Animation on the Macintosh with 3DWorks

John F. Schlag
Julian E. Gómez
Alex D. Seiden
MacroMind, Inc.

Abstract

We describe 3DWorks, a new 3D animation system for the Macintosh personal computer, and some of the user interface tradeoffs involved in its design. The aspects of the interface peculiar to 3D are discussed in terms of standard design principles such as visibility and feedback. We describe the application of outlining and the spreadsheet paradigm to the data management problems of 3D animation. Also described are some attempts at jargon reduction via conceptual unification of traditional techniques. We conclude with some remarks on the similarities and differences between the Macintosh and Unix™ worlds.

Introduction

3DWorks is a 3D animation system for the Macintosh personal computer. This paper describes experience gained from its implementation, which is essentially an attempt to take animation technology formerly accessible only to the "computer graphics priesthood" and make it available to the average Macintosh user.

This strategy depends on several tactics. Paramount among them is the graphical interface. This is more than just icons and mice. The Macintosh is really about direct manipulation. The interface must put as few links as possible between the motions of the user and the motions of the objects. Second, 3DWorks makes use of data management techniques already familiar to the Macintosh community. These include outlining, for managing hierarchical data, and the spreadsheet paradigm, for interacting with numerical, textual, and other parameters. These techniques are used in the *Score* window, which also uses a time-line notation much like a traditional exposure sheet. Third is elimination of superfluous jargon. There is a large body of terms, familiar to computer graphics programmers, that are of little or no interest to animators. The 3DWorks interface provides some conceptual unification of traditional techniques by substituting graphics for jargon.

Although 3DWorks is implemented on a personal computer, and with unsophisticated users in mind, it does not sacrifice functionality. Where the power of the CPU is the limiting factor, such as in drawing, our approach has been to sacrifice speed instead. Stand-ins and interruptible rendering are used to maintain good interaction response.

The Graphical Interface

As mentioned above, a graphical interface is the first priority. In a sense, this is what the Macintosh is all about. There were many challenges in designing a graphical interface to even basic 3D keyframe animation. The key aspects of the interface are [Norman]:

• VISIBILITY - Users should always be able to see what options they have. Therefore, the controls of the system should be visible. Problems arise, however, in supporting a large number of controls (or commands) with limited screen space. Screen clutter slows down experienced users and intimidates inexperienced ones. The solution is a necessary evil: commands are divided into two camps: simple and expert. The primary problem, then, is categorizing the controls. There should be no shame in admitting this to be an iterative process. Some controls, like basic object creation and movement, are immediately visible. Others, such as numerical input, were categorized only after user testing.

It is interesting to consider the basic Macintosh user interface guidelines (and their embodiment in such well known software as MacDraw, MacPaint, etc.) in terms of visibility: some controls are immediately visible, while others are "perusable" in menus. The keyboard shortcuts visible in menus expedite the transition from "knowledge in the world" to "knowledge in the head". Beginners rely on exploring menus and controls (the application's world) to learn how to use a new program. Experts often rely on shortcuts (in their heads) for speed.

• A GOOD CONCEPTUAL MODEL - The user's basic conceptual model of the animation process is: objects are selected; movements are applied to them to determine key frames; the system interpolates the keys to produce in-between frames. There is a current frame, represented visually by a small moving rectangle called the *playback head*. Changes to the selected objects are applied at the current frame.

The model becomes more complicated when users want more control. For example, objects have a number of parameters, and keys can be positioned independently for each parameter. (In this respect, 3DWorks is a *key parameter* rather than a *key frame* system [Gomez84].) The important point is that the model is simple for beginning users. It also becomes more complicated in reasonably linear relation to the amount of independent control desired (through the use of data management techniques described later). Process administration, interprocess communication, and other such details are hidden from the user. Even though the system consists of several cooperating applications, the user sees only one interface.

• DIRECT MANIPULATION AND NATURAL MAPPINGS - The object movement controls, in particular, need to make the user feel that he or she is manipulating the objects in as direct a manner as possible. This requires not only the "point, click, and drag" interface typical of Macintosh applications, but also intelligent relationships between the coordinate system of the mouse, the coordinate system in which the transformation is applied, and the nature of the transformation itself. In translation, for example, an obvious choice might be to relate the x/y coordinate system of the mouse to the local x/y coordinates of the object. This is a mapping which would only be "obvious" to a programmer. What's obvious to a user is that the motion of the mouse should be related to screen space - not object space, world space, or even camera space, but pixel-for-pixel correspondence to cursor movement on the screen. This is especially true for users accustomed to the $2^1/_2$D world of current drawing and animation programs. For advanced users who want more esoteric mappings, there are other choices of coordinate system.

• FEEDBACK - The results of user actions should be immediately apparent. Feedback in 3DWorks is provided primarily through views of the scene, and as continuously as possible. Reality on the Macintosh, however, is that drawing is painfully slow. Therefore, stand-ins such as bounding boxes are used whenever the user is dragging something — often an object, although other things such as the playback head require view updates. Screen drawing is interruptible (in spite of the single process nature of the Macintosh) so that the user can start a new movement or other command without waiting for drawing to finish. The drawing quality can also be tailored by hand, from 2D icon stand-ins, through 3D wireframes and simple shaded renderings to antialiased, textured renderings. Also, note that "wysiwyg" is actually a feedback principle. Fully rendered frames are the same size as preview frames, aspect ratio and device independent.

The one thing the designers of 3DWorks have all agreed upon is that 3D animation is complicated. The best laid graphical interface will never be perfect, nor will one interface please everyone. Hence, 3DWorks employs a typical user interface safety net: the online help system. The help is concise — it is not an online version of the entire manual. It is direct, in that the user can point and click at anything to get help, and the feedback is immediate. The help system recognizes the various details of the interface, rather than just the larger aspects such as windows. There is also a simple hypertext facility for getting to cross references.

One final comment about the graphical interface: a highly interactive Macintosh application is not created by porting a Unix application with a command line interface to the Mac and pasting on a graphical front end. In fact, we tried this strategy, and it became clear early on that the software modules needed to be much more closely coupled to support the level of interaction we wanted. Although 3DWorks is loosely based on Twixt [Gomez84], it is not a port, but a complete rewrite.

Data management techniques

It is important that beginning users see a simple interface, but also that advanced users can obtain detailed control. In particular, users from science, engineering and education may require *exact* control, since they may have some model underlying an animation. This requires the user interface to provide an overview of the data for novice users, and a means of accessing detail for advanced users. 3DWorks mixes two techniques to solve this problem: outlining and spreadsheets. Because of their natural fit to data management problems, both of these techniques are already familiar to the general Macintosh user community.

The data for each object is stored in an array of *tracks*, one for each object parameter. The tracks are arranged in a hierarchy, with the top level representing the whole object, and sub tracks for position, orientation, scale, shading, etc. There are more tracks below this level for individual scalar, boolean, text, or other type values. The tracks for each object are shown together in the *Score* window (see Figure 1). With data in this arrangement, outlining is a natural technique for presenting and hiding it. Clicking on the small diamond to the left of a track name shows or hides the children of the track. This is very similar to the way text outliners work on the Macintosh [Symantec].

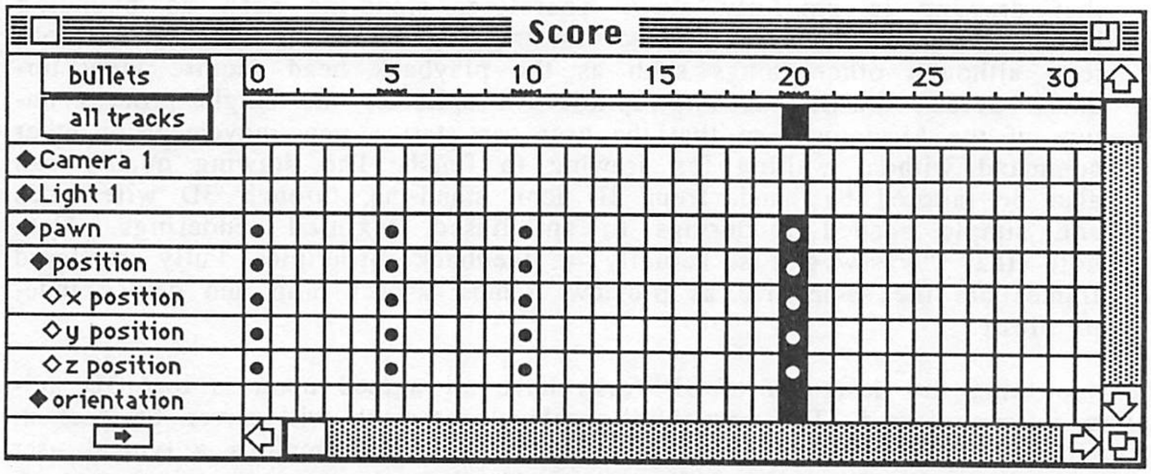


Figure 1: The score window

Time in the score is quantized into cells, which naturally brings to mind a spreadsheet. Each cell contains information about a given track at a given time. The information may be as simple as a bullet, indicating the presence of a key value, or the actual value may be displayed with text and/or color. To edit a value, the user simply clicks on a cell. The vanilla spreadsheet paradigm is extended to include a different editor for each of the basic data types. Several editors simply represent the value as text and invoke the default text editor. In some cases popup menus are used when the set of possible values is small.

The spreadsheet model affords a number of features such as graphical selection, copying and pasting. This is a boon for replicating cycles of animation. Formulas can be used to specify motion procedurally. The formula parser can support extensibility by reading in externally compiled code. This allows advanced users to experiment with new techniques such as dynamics and behaviors.

Another feature of the hierarchical spreadsheet is that data can be edited at multiple levels. The specularity of a surface, for example, can be edited using the scalar text editor on the specularity track, or with a wysiwyg appearance editor on the shading track which shows the effect of the shading parameters on a sample object.

Jargon reduction

One way to eliminate buzzwords is to substitute graphics for text. For example, the user has a choice of interpolation methods to be used between any pair of keys. This amounts to specifying control parameters for splines of various sorts. The bare minimum would be to supply a list of the standard basis functions (Figure 2a), but with more recent formulations [Kochanek, Duff] we can do better (Figure 2b).

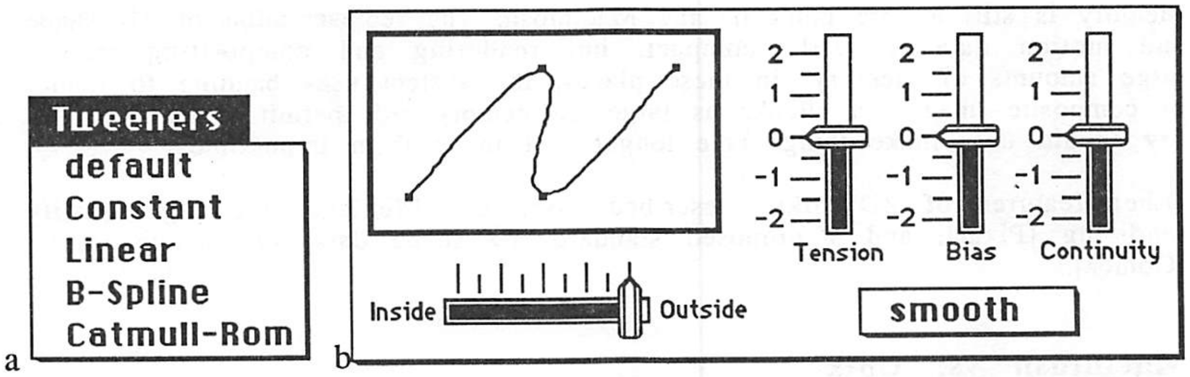


Figure 2: Interpolation interfaces

Another example of the conceptual merging of traditional techniques in 3DWorks is the handling of 3D rotations. Matrices and homogeneous coordinates have been the method of choice for concatenating transformations and transforming points for some time. Euler angles have traditionally been used for interpolating rotations, and quaternions have recently come (back) into favor for the same. 3DWorks uses all three formulations for various purposes: matrices for concatenating movements, quaternions for interpolation, and Euler angles for textual display and editing. The user never needs to know the details of representation and conversion. The basic user interface to rotation is still point-and-click. (Our method is very similar to the virtual sphere [Chen].)

There is a rather subtle jargon-related issue centering around user education. We don't want to rely on users reading text, but it doesn't hurt to educate them along the way. They're most likely going to see the terms elsewhere, if they haven't already. Also, more technical users often need to know exactly what is going on. For this reason, it is sometimes preferable to keep the jargon around, attached to the graphics in some way. For example, the shading terms *flat*, *Gouraud*, and *Phong* are often a stumbling block to newcomers. Recently, however, RenderMan [Pixar, Upstill] has defined the more general and technically orthogonal notions of *shading rate* and *shading interpolation*. We don't want to completely eliminate the original terms from the interface just as users are learning what they mean, but we'd like to educate users to the new ones.

Toys vs. Tools

Our goal was not to put flying block letters into the hands of the masses. We intended from the outset that 3DWorks would provide a user interface and systems support for more advanced animation techniques. The extensibility mentioned earlier allows ambitious users to experiment by loading custom code at run time. We think this will be especially important to academic users.

3DWorks provides three levels of control over parameters, for those with different needs. For appearance-driven animations, motion controls can be used free form via the mouse in the view windows. For fine positioning, the controls can also be used in single-step mode. For those interested in exact positioning or simulation, the score can be used to edit parameters textually.

Memory is still a sore point on the Macintosh. The representation of 3D shape and motion data is fairly compact, but rendering and compositing require large amounts of memory. In these places, the system uses banding to render or composite images in chunks as large as memory will permit. Lack of memory should only make things take longer, not make them impossible.

Other features of 3DWorks (described elsewhere [Schlag]) are high quality rendering [Pixar], and a proposed standard 3D shape data interchange format [Gomez].

Macintosh vs. Unix

The traditional view of animation is the pipeline: modeling, animation and rendering. In practice, this view is inadequate. A better view is the toolbox view, in which users are given a collection of atomic operators which they must assemble to do a job. The Macintosh OS and Unix do not differ on this point: both provide the necessary data transport mechanisms. The difference is the means by which the user must assemble data. On the Macintosh, everything is done interactively, and graphically, where possible. This is a selling point for technophobes. This model breaks down, however, when a task must be repeated many times, which is of particular importance in animation. The feature of Unix which really makes the toolbox approach work is the shell [Bourne], which allows users to extend the system via scripting. On the Macintosh, scripting is the baby which was thrown out with the bath water during development. This is only now being corrected.

The problem with scripting for beginning users is that it is indirect. Also, since script commands are not visible to the user, it is difficult to figure out what sequence of commands is necessary. One solution to this is scripting by example: the user interacts graphically, and the system stores the sequence of commands for later use. 3DWorks uses this technique in the specification of image domain special effects to be run for each frame during rendering. The script is stored as text, and can be edited by the user.

Toward the future

We would like to explore the use of outlining for time as well as for object data. An animator should be able to view a breakdown of a scene by action, rather than by key parameters. Parametrization of animation is a related issue, which would enable animators to create motion which could be applied to any object or collection of objects.

In a large interactive system such as 3DWorks, the user interface is 95% of the work. It doesn't have to be this way. Some of the user interface prototyping work for 3DWorks was done with MacroMind Director Interactive [MacroMind]. This allowed an animator to produce a mockup of an interface with the "look and feel" of the final product. The appearance elements of the mockup (menus, icons, controls, etc.) could then be used in the implementation. We plan to streamline this process even more. The interface designer will be able to incorporate interpreted scripts to simulate more of the behavior of the program. The programmer will then fill in and/or replace scripts with compiled code where extra functionality and/or efficiency are required. The mockup can grow into a complete application.

An exciting prospect for user interfaces to 3D graphics is iconic programming. At first glance [Glinert], the visual programming literature seems rather negative. All the experience with visual programming seems to indicate that it is intractable for large projects. On the other hand, the positive results have come from small, well-defined application areas. Fortunately, there are several places in 3D graphics that fit this description. Icons and connections look like a good way to provide the generality of programming to "non-programming" users. (One of the strange paradoxes about the general user community is that its members are often willing to program, but only if the programming is disguised as something else. Spreadsheet models and scripting by example are both cases of this. Also, the success of Hypercard [Goodman] indicates that the disguise may be exceedingly thin.)

We would like to design a user configurable input device layer that allows new devices to be connected and used without the addition of new software. The main reason for this is that manipulating 3D objects with a 2D mouse is tedious, at best, but a cheap 3D input device has yet to be developed.

Acknowledgements

The authors would like to thank Stuart Sharpe, for user interface design and mockups; Erik Neumann, Al McNeil, Galyn Susman, Nancy Tague, and John Lasseter for additional ideas; and finally Marc Canter, for supporting a skunkworks while most of MacroMind's attention was focused elsewhere. Also, Macintosh applications typically owe a debt to their predecessors. Therefore, the authors would like to thank the creators of existing applications for contributing to the success of graphics on the Mac.

References

Bourne, S., *The Unix™ Shell*, Unix Programmer's Manual, Volume II.

Chen, Michael, S. Joy Mountford and Abigail Sellen, *A Study in Interactive 3-D Rotation Using 2-D Control Devices*, Proc. Siggraph 1988.

Duff, Tom, *Families of Local Matrix Splines*, in *The Mathematics of Computer Graphics*, Course Notes #15, Siggraph 1984.

Glinert, Ephraim, et al., *Introduction to Visual Programming Environments*, Course Notes #17, Siggraph 1989.

Gomez, Julian E., *Twixt: A 3D Animation System*, Proc. EuroGraphics 1984.

Gomez, Julian E., John F. Schlag and Alex Seiden, *3DGF, Version 1.0*, MacroMind, Inc., San Francisco, CA.

Goodman, Danny, *The Complete HyperCard Handbook, 2nd Edition*, Bantam Computer Books, 1988.

Kochanek, Doris H. U. and Richard H. Bartels, *Interpolating Splines with Local Tension, Continuity, and Bias Control*, Proc. Siggraph 1984.

MacroMind, Inc., *Director Interactive Toolkit Manual*, MacroMind, Inc., San Francisco, CA.

Norman, Donald A., *The Psychology of Everyday Things*, Basic Books, Inc., NY, NY, 1988.

Pixar, *The RenderMan Interface, Version 3.0*, Pixar, San Rafael, CA.

Schlag, John F. and Julian E. Gomez, *3DWorks: A New 3D Animation System*, in *Desktop Computer Animation*, Course Notes #3, Siggraph 1989.

Symantec Corp., *More II Reference*, Cupertino, CA.

Upstill, Steve, *The RenderMan™ Companion*, Addison-Wesley, 1989.

Winer, Dave, *About UserLand IPC*, UserLand Software, Inc., Mountain View, CA.

The Acorn Outline Font Manager

Neil Raine, David Seal, William Stoye, Roger Wilson

Acorn Computers Ltd, Cambridge Technopark,
645 Newmarket Road, Cambridge CB5 8PB, England.

Acorn's Archimedes personal computer uses an outline font manager to provide fonts on screen and on printers, from a single outline source. The algorithms used for font scaling, and conversion to bitmaps, are an updated version of those described in [Hersch 87]. In addition to this, anti-aliasing is used to improve the quality of fonts on-screen [Warnock 80].

This paper describes some of the results of using these techniques together in a practical environment. First the overall structure of the software is described, then the details of some of the algorithms employed, and finally some observations on the use of this system by a wide user base. The system has been on sale in the UK since about May of 1989 ([PCW] contains a review of this product) and already has several thousand users.

System structure

The fonts are held on disc in files that describe the outlines, and overall metrics, of each individual glyph. When a specific font/size is requested by an application it is loaded into memory and the required characters are converted into bitmaps. The bitmaps and outlines are held in a cache, so that reconversion to bitmap (and reloading from disc) are minimised.

Four main different rendition techniques are used by the font manager. The choice is made based on the size of the characters being rendered, with the actual values of division points being configurable by the user.

For the biggest characters, the outlines are drawn directly to the destination (either the screen, or a bitmap being prepared for printing). This prevents the cache from being filled by rarely used enormous characters.

For the next size down, the outlines are drawn into bitmaps, which are retained in the cache. This provides identical results to the first method, but is much faster when a character is drawn more than once.

For the next size down, the outline is drawn into a bitmap four times too big, and then anti-aliased down to a 4-bit-per-pixel bitmap of the correct size. This bitmap is kept in the cache.

For the smallest characters, four different anti-aliased versions of each character are built, based on the horizontal placement of the character within the pixel grid. This smooths out the placing of closely packed characters, causing a considerable improvement in characters that are only a few pixels across.

Grid constraints

Hersch's techniques centre around the addition of *grid constraint* information to character outlines. This grid constraint information is used to move the elements of the outline by small amounts, before it is filled in, dependent on the alignment of the glyph with the destination raster.

Some of the differences between Hersch's techniques and our own are quite significant, so we explain the method in full rather than differences from Hersch's work.

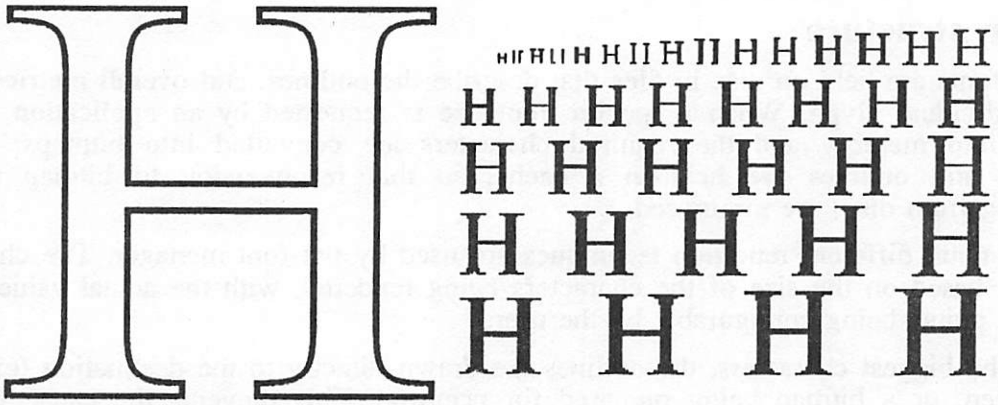
Simple Outlines

A character glyph is held as a path, consisting of the following elements:

```
moveto(x, y)
drawto(x, y)
curveto(x1, y1, x2, y2, x, y)
closepath
```

The coordinate system is expressed in underlying absolute design units, typically thousandths of an em in the current font. The moveto elements divide the path into subpaths. Each subpath starts with a moveto, and optionally ends with a closepath. The drawto is a straight line edge in a subpath, a curveto is a Bezier curve with two control points defined by (x1, y1) and (x2, y2).

The glyph is rendered by filling in the above path using the even-odd fill rule, familiar to PostScript programmers [PostScript 85]. The path is calculated to high resolution, following which a pixel is set if its centre lies within the path.



The diagram shows the outline of a typical letter, and some typical results if one depends entirely on these algorithms. The results are shown in various sizes from 6 point to 50 point, at 90 pixels per inch. Note how some strokes of the letter can get narrower at larger sizes, that the serifs at any particular size do not match (even though they are identical in the outline), and that the vertical strokes can be of different widths.

Line Grid Constraints

The most important change needed to the method above is to improve the handling of vertical and horizontal strokes in the glyph. Vertical and horizontal line grid constraints are designed to achieve this.

A vertical line grid constraint consists of:

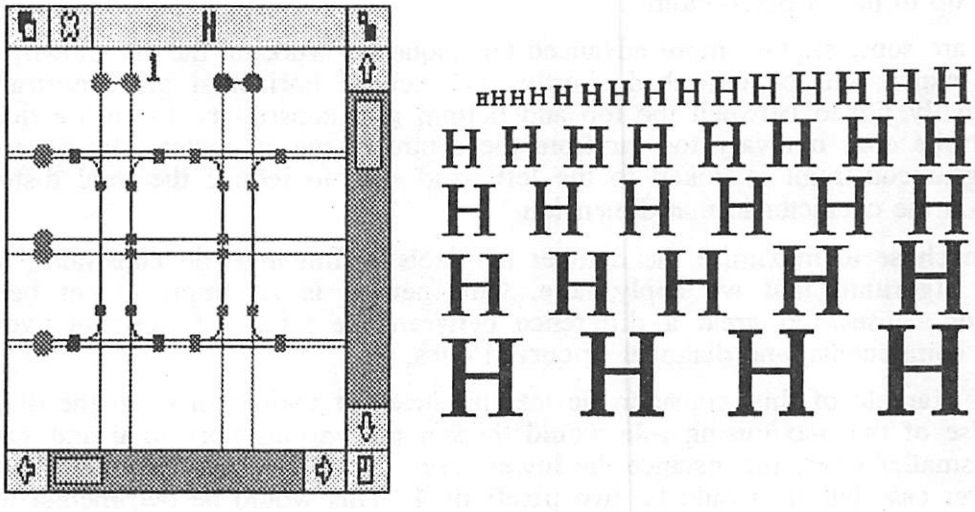
an x coordinate within a glyph's coordinate space
a width
a set of points on the path of the glyph which are linked to this grid constraint

The constraint should be thought of as corresponding to a strong vertical component of the glyph to which it applies.

When rendering a glyph to the output device, the grid constraint is moved in the x direction by a small amount (half a pixel or less). All points on the glyph path which are linked to this grid constraint have their x coordinates moved by a similar amount.

The exact movement is such that the number of pixel centres within the grid constraint is a constant function of the exact width of the constraint. The ideal width in pixels is calculated, and if the fractional part of this is greater than a quarter (or if the integer part is 0) then the width is rounded up. Then, the grid constraint is moved by the minimum distance which will cause it to cover that many pixel centres.

A horizontal line grid constraint has precisely analogous properties and rules, but swapping "x" and "y", "vertical" and "horizontal" in the description above. A point may be linked to one vertical constraint (which moves it horizontally) and one horizontal constraint (which moves it vertically).



The diagram shows a screen dump from the grid constraint editor program, which is used to add grid constraint information to a font outline. A horizontal or vertical grid constraint is represented on the screen as a pair of lines, showing the direction and width of the constraint. The diagram shows two vertical grid constraints and three horizontal ones as applied to a Times Roman H. The points on the outline of the character are linked to the grid lines, as follows:

All the points in the left half of the character are linked to the left hand vertical constraint. This ensures that the left hand vertical stroke has desirable properties.

All the points in the right half of the character are linked to the right hand vertical constraint. This ensures that the right hand stroke has the same properties (and the same width) as the left hand vertical stroke.

All the points at the top of the character are linked to the upper horizontal constraint.

The four points on the central stroke are linked to the middle horizontal constraint.

All the points at the bottom of the character are linked to the lower horizontal constraint.

The diagram also shows the resulting generated bitmaps at various point sizes between 6 point and 50 point, at 90 pixels per inch.

The grid constraints ensure that the vertical strokes have the same width in the destination (they have precisely the same width in the source outline). They never disappear at tiny sizes, and their width increases monotonically with increasing point size. The width will also be identical to the width of similar strokes in other letters (such as I, L, E) rendered at the same point size. The central horizontal stroke has similar properties.

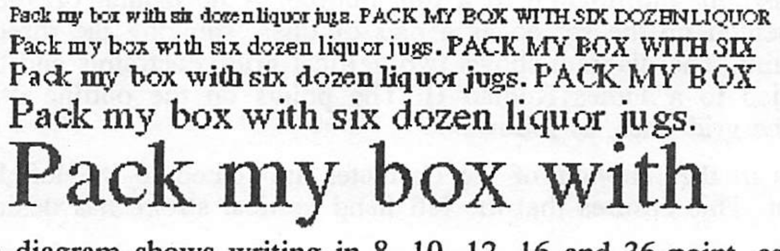
The four serifs are all identical in the resulting bitmaps, because of the adjustment made in both dimensions. They never disappear, and match similar serifs in other characters.

The grid constraints are moved independently. Each one can move portions of the glyph by up to half a pixel-width.

There are some slightly more advanced techniques at work on the Hs shown in the diagram, which will be described shortly. The central horizontal grid constraint is proportionally linked between the top and bottom grid constraints, to ensure that the central stroke does not vary too far from the centre of the character. The right hand vertical grid constraint is linked to the left hand one, to reduce the total distortion possible on the character in that dimension.

Hersch chose to maximise the number of pixels within the grid constraint, rather than the algorithm that we apply here. Our method is an improvement because maximising causes too great a difference between the width of horizontal/vertical character components, and diagonal or curved ones.

As an example of this, consider the text produced at various sizes in the diagram below. Use of the maximising rule would thicken the various horizontal and vertical stems at smaller sizes, for instance the lower case l in 10 point (and similar stems on other lower case letters) would be two pixels thick. This would be detrimental to the appearance of those letters.


Pack my box with six dozen liquor jugs. PACK MY BOX WITH SIX DOZEN LIQUOR
Pack my box with six dozen liquor jugs. PACK MY BOX WITH SIX
Pack my box with six dozen liquor jugs. PACK MY BOX
Pack my box with six dozen liquor jugs.
Pack my box with

The diagram shows writing in 8, 10, 12, 16 and 36 point, assuming 90 pixels per inch, using all of the techniques described in this paper. Note how important the relative heights of letters are, including letters whose base lines do not in fact precisely match (such as an x and an o). Note also how the widths of specific strokes increase in a systematic and coherent way, and that the behaviour of serifs is consistant at any particular size.

Arc Grid Constraints

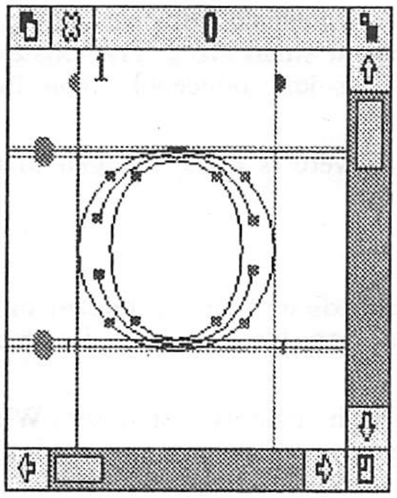
As curved portions of a glyph cross the horizontal or vertical, a single pixel (a "pip") or a long straight run of pixels is immediately noticed as degrading the smoothness of the curve. The line grid constraints are designed to help horizontal and vertical lines, and are of little use in preventing this.

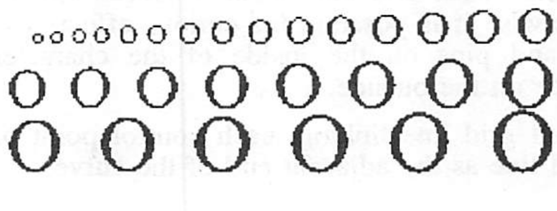
A left or right arc grid constraint consists of:

an x coordinate
a set of points on the path of the glyph which are linked to this grid constraint.

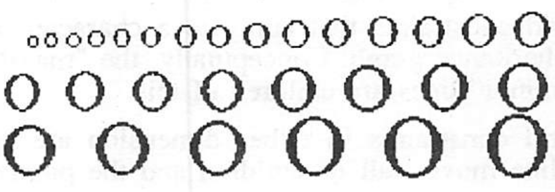
A top or bottom grid constraint consists of:

a y coordinate
a set of points on the path of the glyph which are linked to this grid constraint.





Without grid constraints



With grid constraints

The diagram shows a letter that includes constraints of this form, as it appears within the grid constraint editor. The constraints are represented by a vertical line at the given coordinate, with a semi-circular "handle" on it to indicate the direction of curvature. The diagram above shows a Times Roman O with four different arc grid constraints, one of each type. The points on the outline of the character are linked to these as follows:

All the points on the left half of the character are linked to the left arc grid constraint (on the left).

All the points on the right half of the character are linked to the right arc grid constraint.

All the points on the upper half of the character are linked to the top arc grid constraint.

All the points on the lower half of the character are linked to the bottom arc grid constraint.

When rendering a glyph to the output device, a right arc grid constraint moves the line to 7/8ths of a pixel from the left boundary of a pixel, with the cutoff point at 1/2 of a pixel. Thus, if the initial position is x, then the new position x' is

```
x' = INT(x-1/2) + 7/8
```

A point can be linked to at most one vertical, left arc or right arc grid constraint.

Left, top and bottom arc grid constraints behave in an exactly analogous manner in the y direction.

The diagram above shows the effect of these contraints on the letter O, displayed in point sizes ranging from 6 point to 34 point.

When constructing long continuous curves, the decision about where to split the curve into Beziers can be quite important, because it is important to preserve the overall symmetry of the character. The O is split into four Bezier curves at 45-degree angles, as this makes it clear which points on the path should be linked to which lines.

Note that the O in the diagram has two internal lines at the top and bottom of the character. These are skeleton lines, see below: they prevent the line from entirely disappearing at small sizes.

Both the inside and outside of a curved stroke should be linked to the same grid line: otherwise it is possible for curious effects to occur at small sizes. The existence of runs and pips on the inside of the character is far less noticeable than their appearance on the outside.

With all grid line linking, each control point of a curveto is always linked to the same grid line as the adjacent end of the curve.

Related Grid Constraints

The grid constraints that apply to a character in each dimension are related in an acyclic inheritance graph. Conceptually, the "major" lines are the parents of the graph and the "minor" lines are children of this.

The grid constraints in either dimension are applied in parents-first order. When any grid line moves, all its children and the points linked to them move by a similar amount.

This structure is needed to cope with cases in which grid lines are very close, and at low resolution must become coincident rather than differing by a pixel. For instance, consider the base line of Times Roman: the bottom of an o should descend slightly below the feet of an x. However, at low resolution these must become coincident. This can be achieved by relating the bottom-of-o line to the bottom-of-x line: the precise algorithms chosen above mean that the two will not be chosen as different values, unless the difference between the lines is more than half a pixel.

Another example is the loops of a B: the lower one is slightly larger than the upper one, but at low resolution they should be coincident rather than the lower one sticking out by an extra pixel.

Yet another example is the loops of an m. There are three major vertical grid constraints in an m, for the left, middle and right vertical strokes. By relating middle to left, and right to middle, the loops can be forced to precisely identical sizes.

Proportional Grid Constraints

A grid constraint with a parent and grandparent may be marked as moving "proportionally" between the two. This means that the position of the grid constraint, relative to its parent and grandparent, will be preserved.

An example of the requirement for this is the middle stroke of an E, which is slightly above the middle of the character. Without proportional grid constraints, at some sizes the middle stroke ends up below the centre of the character. The middle of an H is similar.

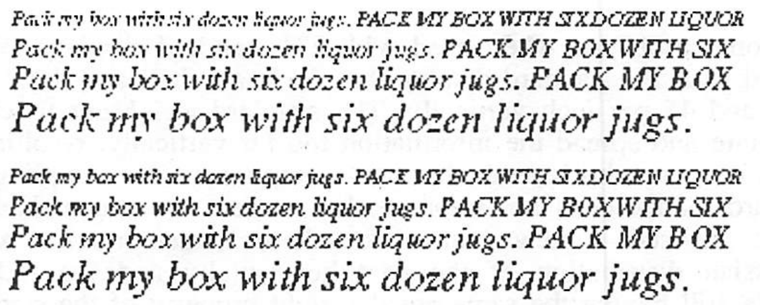
The Grid Hierarchy

Grid constraint lines can be shared between characters, so that the features of one character can affect the shape of another. The characters are related in an acyclic graph. Each character can inherit grid lines from its parent.

Strictly this mechanism is not required, and could be duplicated by placing grid constraints that simply happen to correspond to the features of another character. However, describing and implementing it like this more clearly demonstrates the true relationships between characters. It also makes the representations of grid constraints considerably more compact.

Skeleton lines

Usually, the glyph consists only of closed subpaths. An open subpath is known as a skeleton line, and is traversed at one pixel width. This prevents parts of the outline from entirely disappearing at small sizes, especially thin diagonal strokes.



Pack my box with six dozen liquor jugs. PACK MY BOX WITH SIX DOZEN LIQUOR
Pack my box with six dozen liquor jugs. PACK MY BOX WITH SIX
Pack my box with six dozen liquor jugs. PACK MY B OX
Pack my box with six dozen liquor jugs.
Pack my box with six dozen liquor jugs. PACK MY BOX WITH SIX DOZEN LIQUOR
Pack my box with six dozen liquor jugs. PACK MY BOX WITH SIX
Pack my box with six dozen liquor jugs. PACK MY B OX
Pack my box with six dozen liquor jugs.

In the diagram above the upper text is drawn without skeleton lines, the lower text is identical but with the skeleton lines included. An italic font is used because they are particularly susceptible to the problems that skeleton lines solve.

Discussion

Although our work was originally inspired by that of Hersch, our methods are different from his in a number of significant respects. Hersch produces the original ideas of grid constraints and of arc grid constraints, in order to solve most of the problems discussed above. However, he does not provide the flexibility that the linked and proportional grid line relationships provide. Also, he attempts to maximise the number of pixel centres within grid constraints leading to some of the problems described above.

Anti-Aliasing

Anti-aliasing is used to improve the quality of text on the screen. A straightforward approach is used to construct the anti-aliased information: the character is rendered into a bitmap four times over size (using all the techniques discussed above) and then pixel collections are sampled to produce an output pixel.

The standard weighting functions [Warnock 80, Crow 78, 81] were tried initially. Of these, the following 7 by 7 weight matrix gave the best results on monitors with square pixels:

```
1  2  3  4  3  2  1
2  4  6  8  6  4  2
3  6  9 12  9  6  3
4  8 12 16 12  8  4
3  6  9 12  9  6  3
2  4  6  8  6  4  2
1  2  3  4  3  2  1
```

The numbers in the matrix give the weight for each pixel of the source to contribute to the final value of the anti-aliased pixel. The matrix is positioned so that the central 16 is at the centre of the output pixel. It is then moved right/up by 4 source pixels to calculate the next output pixel. Each pixel of the source contributes the same weight to the output display (16) - a very important characteristic for an even display quality.

The computation of the final pixel weight can be done by conditionally adding the values in the matrix depending on the source pixel being on or off. An important improvement on this is to regard the source pixels of each row as a binary number, and to perform the computation for each row as a single lookup into a pre-computed table.

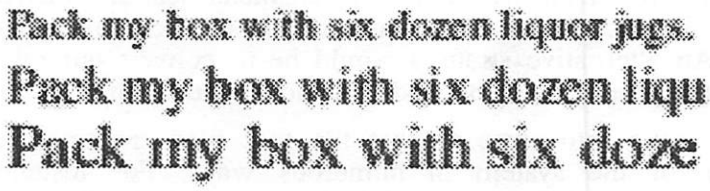
The matrix exhibits some problems when used with TV standard displays. When the computer is interfaced to a TV standard display the pixels are "tall and thin" with 90 per inch horizontally and 45 per inch vertically. The standard weighting functions assume the pixels are square and spread the information too far vertically, resulting in a very fuzzy appearance on some horizontal character components. Re-sampling the data after using a standard matrix does not improve the image - too much data has been lost. So we have devised a new matrix for these screen modes which corresponds to the gaussian distribution of the spot horizontally and the reduced resolution vertically, while still having the same equal weight property of the standard matrix:

```
0  0  0  0  1  0  0  0  0
1  2  4  6  6  6  4  2  1
1  4  8 12 13 12  8  4  1
1  4  8 12 14 12  8  4  1
1  4  8 12 13 12  8  4  1
1  2  4  6  6  6  4  2  1
0  0  0  0  1  0  0  0  0
```

Note the "shape" of the matrix and the greater size horizontally combined with a smaller weighting for the vertical information.

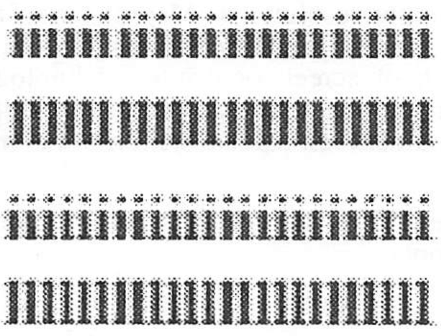
Indeed, even with square pixels, we discovered the new matrix produces slightly superior results: the adjacent pixels horizontally do tend to merge with each other, while remaining more distinct vertically. The row lookup scheme described above means that the slightly larger matrix requires no more computation than the smaller one.

The characters are stored with four bits per pixel of the grey scale weight (the 0-256 value produced by the matrix being limited to 0-255 and truncated to the four top bits). A dynamically computed table (which can be specified by the programmer) maps these four bits into the colour range on the display. Typically (when running the multi-window, multi-application desktop) only 3 bits per pixel are actually used to display the character, but the quality loss is minor. Coloured characters can also be anti-aliased with appropriate setting of the table. When the computer is using an 8 bit per pixel display, colour anti-aliased characters are also convincing (and in this state all 16 grey levels are used for normal white/black text). A drawback with anti-aliased text is that the background colour (the colour the characters are painted over) must be specified in order for the anti-aliased levels to be computed.


Pack my box with six dozen liquor jugs.
Pack my box with six dozen liqu
Pack my box with six doze

The diagram shows text in 8, 10 and 12 point, magnified by a factor of two so that the grey scales are apparent. Anti-aliased text on paper looks dreadful! The actual effect on the viewer can only be understood by seeing the results on-screen.

To increase the positioning resolution, sub pixel anti-aliasing can be used. The character is rendered four times over size as before, but four positions are chosen (offsets 0, 1, 2 and 3) in order to compute a set of different anti-aliased versions of the character. The appropriate sub-pixel position is chosen when the character is painted, resulting in a greater fidelity for the original text. This option is particularly appropriate for small typefaces - which is fortunate since the preparation of a sub pixel anti-aliased group of characters uses 4 to 16 times (horizontal only or both horizontal and vertical) as much processing time. A user programmable threshold automatically controls this option, turning it on when the point size is small enough. Generally most of the quality improvement is found in the horizontal direction.



The diagram shows some text in 12 point, magnified by a factor of two so that the grey scales are apparent. The upper two lines consist of "i"s and "l"s rendered using anti-aliasing: positioning irregularities are very obvious, and lead to an additional blank pixel every few lines. The lower two lines use sub-pixel anti-aliasing, resulting in a smoothing out of this effect.

This example is chosen because it highlights the differences most vividly. The effect on the eye is only fully apparent when viewed on-screen, and is particularly important with very small point sizes.

Other techniques

A variety of other techniques and refinements is required in order to make this system work in a practical environment.

The algorithms above are used to print to a variety of printers such as LaserJets and dot-matrix printers, with a variety of resolutions. With a PostScript printer, however, the operating system's font-painting primitives are intercepted and replaced by textual PostScript that uses the printer's own built-in fonts. On doing this with any external font system there are problems with matching character sets and metrics. We solve these problems for PostScript by using fonts with metrics that match the standard PostScript ones. An alternative strategy would be to convert our own fonts into PostScript ones (with grid constraints encoded) and download them.

Considerable effort has gone into ensuring that the font files are small, as this improves the performance of the system in numerous ways. For instance, the following figures are for fonts containing the 192 characters of the ISO standard Latin-1 character set, and 17 other miscellaneous characters:

Times: 29868 bytes
Courier: 31312 bytes
Helvetica: 21160 bytes

This includes all metric information. Cache space taken up by the same outline information is approximately the same.

Another technique that we use is to save the anti-aliased bitmap forms of commonly used fonts in files. This can save conversion time, particularly when sub-pixel anti-aliasing is used.

Observations

The advantages of having a single, scalable description for a font are fairly obvious. It can be used for a wide variety of output resolutions. Magnification can be applied to the screen display or printed output, and the fonts do not have to be tuned to the precise resolution of a specific generation of screen or printer technology. By using the same source for screen and for printer, the user knows that the specific glyphs or point sizes used in preparing a document on-screen will not be missing when the document is printed.

Here are some typical performance figures. The figures give the number of characters rendered per second from a mix of fonts.

Point size, on a 90dpi screen:	8 point	12 point	18 point
Render from cache in monochrome:	3970	2880	2173
Render from cache, anti-aliased:	4430	3630	2660
Read and convert from disc in monochrome:	173	156	141
Read and convert from disc, anti-aliased:	106	91	76

Horizontal sub-pixel anti-aliasing quadruples the conversion time. The differences between rendition times using the different methods reflect different levels of careful optimisation in inner loops. The measurements were made on an ARM3 processor, running at around 10 MIPs [Furber 89, Furber 89a].

The grid constraint editor has been used successfully by font designers and graphic artists, who took several months to gain an understanding of the algorithms above, but can now add grid constraints to a font in about 2-3 man-days, once suitable outlines have been prepared.

User response to anti-aliased fonts is varied. If screen resolution is less than around 50 dots per inch in either dimension then anti-aliased fonts are not popular for all-day use in editing mail messages and programs, a simple monospaced system font being preferred. Above this, however, the anti-aliasing is extremely popular. Many users with such high-resolution monitors are happy to edit everyday text and mail messages using 8 point typefaces on the screen. At any resolution, anti-aliasing is found to be a distinct advantage when previewing typeset text on the screen. Much useful information can be gained by previewing with all units reduced by a factor of two: text is legible (but not easily readable) down to 4-5 point. Quality of italic typefaces and the preservation of the weight of a typeface are particularly noticeable compared to a one bit per pixel version of the same typefaces.

There is no doubt that scalable font technology will become an essential part of all personal computers. The methods described in this paper have provided excellent results, and a reasonable tradeoff between system complexity and resulting performance.

The place of anti-aliasing is less clear. Its use has not yet caught on widely within the industry, but as desktop processor power gets cheaper (and colour displays more common) this may change. In our experience, anti-aliasing is well worth the run-time costs involved.

References

[Crow 78]:
The Use of Grayscale for Improved Raster Display of Vectors and Characters,
SIGGRAPH '78 Proceedings. (Computer Graphics 12, August 1978)

[Crow 81]:
A Comparison of AntiAliasing Techniques,
IEEE Computer Graphics and Applications, January 1981.

[Furber 89]:
VLSI RISC Architecture and Organization,
S. B. Furber,
Marcel Dekker Inc., New York, 1989 (Chapter 5).

[Furber 89a]:
ARM3 - A 32b RISC Processor with 4k Byte On-Chip Cache,
S. B. Furber, A. R. P. Thomas, H. E. Oldham, D. W. Howard, J. S. Urquhart and A. R. Wilson,
VLSI '89 conference proceedings, Elsevier, 1989.

[PostScript 85]:
PostScript language reference manual,
Adobe Systems Incorporated,
Addison Wesley 1985.

[Hersch 87]:
Character Generation under Grid Constraints,
Roger D. Hersch,
ACM SIGGRAPH Volume 21 Number 4 July 87, pp 243-252,
SIGGRAPH '87 Conference Proceedings from July 27-31 in Anaheim, Ca.

[PCW]:
Roger Howarth,
Acorn DTP,
Personal Computer World, July '89, pp 178-181.

[Warnock 80]:
The display of characters using gray level sample arrays,
John E. Warnock,
Computer Graphics, 14(3), pp. 302-307, July 1980.

NeWS Classes; an Update

Owen Densmore
Sun Microsystems
Mountain View, CA 94043
(415) 336-1798
odensmore@eng.sun.com

The use of classes by the NeWS 1.0 toolkit "Lite" was introduced at the 1986 Monterey Graphics Conference as a tentative exploration of packaging for PostScript programs residing in the NeWS Window System. The NeWS Toolkit (TNT) used by the X11/NeWS Window System has greatly extended the use of classes over its precursor. This is a report on these activities.

This paper is divided into three sections. The first is on the basic PostScript class model used in NeWS. It discusses the fundamental notions of PostScript dictionaries as a means for inheritance, objects as dictionaries, how messages are sent, sending to self and super, creation of classes and instances, and the basic class Object.

The second section continues into advanced topics, many of which are new in the X11/NeWS release. These include multiple inheritance, local variables for methods, redefining instance variables, promotion of class variables to instance variables, obsolescence, re-definition of classes, dynamic creation of methods, defaults, and much more. These topics arise from the far greater use of classes by The NeWS Toolkit than its Lite precursor.

The final section looks at applications of classes in The NeWS Toolkit; applying the advanced topics of the second section. These include automatic destruction of obsolete objects, "magic" dictionaries as instances, use of the canvas tree to form a container hierarchy, implementation of an X window manager in TNT, aids in callback management, a look-and-feel independent User Interface abstraction, and more.

1.0 The Basic Class Model

The PostScript dictionary object can be used to implement Smalltalk-80[5] classes and objects. This section presents the dictionary model for objects and the primitives needed to support basic classes in NeWS.

1.1 Introduction to PostScript Dictionaries

The PostScript language[1,2] supports a *dictionary*, an associative lookup table binding a *key* to a *value*. All name references within PostScript are resolved through the current dictionary stack; an ordered array of dictionaries. The lookup proceeds from the most recent dictionary put on the dictionary stack to the initial dictionary. The initial pair of these dictionaries are special: the systemdict contains all system operators and data, and the userdict contains data global to the current application.

The initial dictionary stack for a NeWS application looks like:

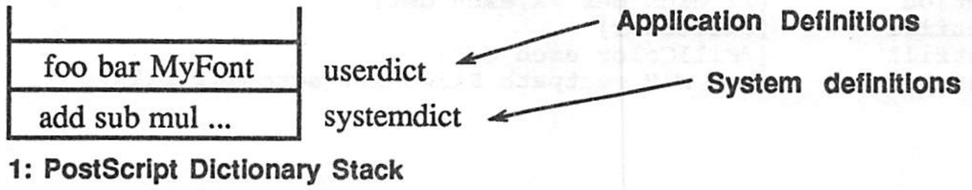


Figure 1: PostScript Dictionary Stack

The key feature of the dictionary stack for object oriented programming is that it provides for name overriding; thereby yielding inheritance.

Name overriding works as follows. Names are looked up from the outermost dict to systemdict. If a system name is found before systemdict, it is used instead. The overriding name can subsequently refer to the system dict object directly by using the **get** operator rather than looking it up by name.

Suppose we want to override the systemdict **add** operator to round down the two values being added. By placing:

```
add { % num1 num2
    floor exch floor exch       % round args down
    systemdict /add get exec    % add them via system operator
def
```

in userdict, we redefine the **add** for the current application.

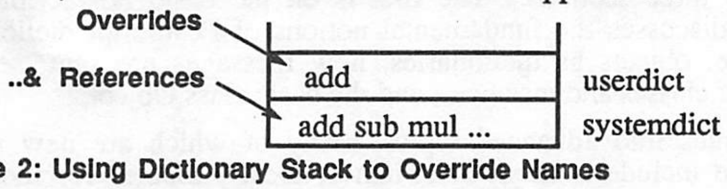


Figure 2: Using Dictionary Stack to Override Names

In object oriented programming parlance, this is an example of the "method" **add** in the "object" userdict invoking the **add** method in its "superclass" systemdict! The object userdict "inherits" the operators **floor, exch,** .. from systemdict when the lookup fails in userdict.

NeWS classes[11] generalize this notion of overriding and inheritance into a Smalltalk-80 style of object oriented programming. This is entirely implemented in roughly 500 lines of PostScript, residing in the server initialization file class.ps.

1.2 What is an Object?

PostScript Objects are an ordered collection of dictionaries treated as a single entity. These dictionaries contain all the data (Class Variables and Instance Variables) and procedures (Methods) needed by the object.

Let's look an a simple rectangle object. It is composed of two dictionaries: the **ClassRect** dictionary that contains the data and methods used by all rectangles, and the **aRectInstance** dictionary that is an instance of a single rectangle whose current location is (100,200) and whose width and height are 50 and 25. The class will contain a single variable, **FillColor**, which is global to all rectangles, and a set of methods specific to rectangles for getting and setting the rectangle location, size and color; and for painting the rectangle.

```
ClassRect:
    /SuperClasses   []
    /FillColor      <red>
    /getsize        {W H}
    /setsize        {/H exch def /W exch def}
    /getloc         {X Y}
    /setloc         {/Y exch def /X exch def}
    /getfill        {FillColor}
    /setfill        {/FillColor exch def}
    /paint          {X Y W H rectpath FillColor setcolor fill}
```

```
aRectInstance:
    /SuperClasses   [ClassRect]
    /X              100
    /Y              200
    /W              50
    /H              25
```

When **aRectInstance** is in use, the dictstack will have the four dictionaries **aRectInstance, ClassRect,** userdict and systemdict

When **getsize** is executed for this object, the names are found in these dicts:

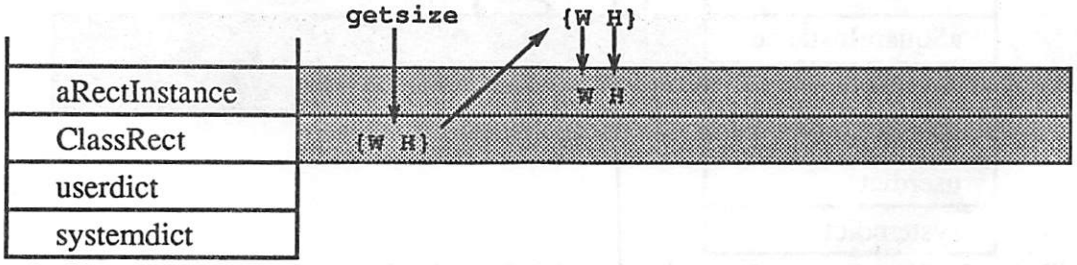


Figure 3: Class and Instance Name Lookup

The method **getsize** is found in **ClassRect.** It executes a procedure looking up the two names **W** & **H** which are found in **aRectInstance.**

The variables **X, Y, W** and **H** are called instance variables; they vary among instances of the class. The variable **FillColor** is called a class variable; it is global to all instances of the class. The procedures **getsize, setsize,** ... are called methods of the class.

Note that classes and instances are both types of objects; they can be put on the dictstack and manipulated. For example, the global class variable **FillColor** can be changed for the class by executing the method **setfill** while **ClassRect** is in use:

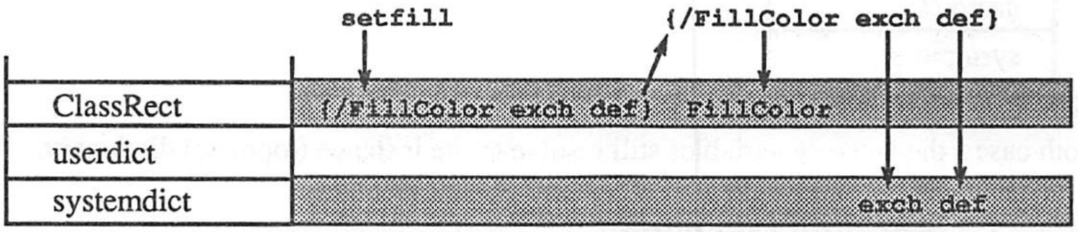


Figure 4: ClassRect Name Lookup for "setfill"

Here the object **ClassRect** is put on the stack, the method setfill is executed and is resolved to the **ClassRect** dict. The procedure executes and uses two names resolved in systemdict which change the value of the **FillColor** class variable. This will change, dynamically, the color for all rectangles immediately.

1.3 Multiple classes: SubClasses

The above example shows only a single dictionary for the class of an instance. Generally an instance has several classes associated with it, each additional class dictionary being a subclass of the initial class.

Let's consider a new class, **ClassSquare,** which adds two new methods to **ClassRect** for setting and getting the "edge" of the square.

```
ClassSquare:
    /SuperClasses   [ClassRect]
    /setedge        {dup /H exch def /W exch def}
    /getedge        {H}
```

```
aSquareInstance:
    /SuperClasses    [ClassSquare ClassRect]
    /X               100
    /Y               200
    /W               50
    /H               50
```

When **getedge** or **setedge** is executed for squares, the method name lookups resolve to the new class, **ClassSquare:**

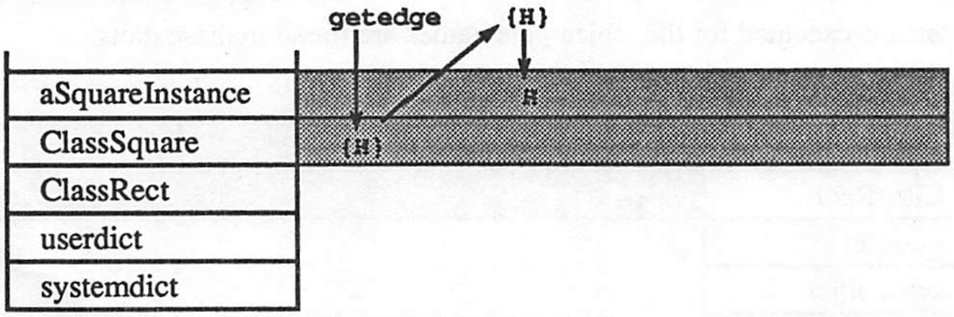


Figure 5: ClassSquare Name Lookup for "getedge"

.But when **getsize** is executed for this object, the method name lookups resolve to the initial class, **ClassRect**:

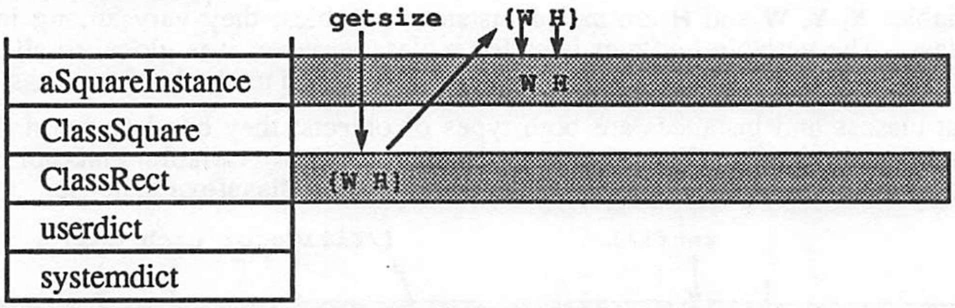


Figure 6: ClassSquare Name Lookup for "getsize"

In both cases, the instance variables still resolve to the instance (topmost) dictionary.

1.4 Sending Messages

Thus far we have not mentioned how objects are taken on and off the dictstack; and how the methods are executed. This is called sending a message to an object and is handled by the **send** primitive.

A message is an optional set of arguments, followed by a method name, followed by the object being sent to, followed by **send:**

```
            arg1 .. argN /method obj send
```

The method may return results by leaving them on the operand stack.

The steps to sending a message are:

- pop off the current object's dictionaries from the dictstack if there is an object currently in effect.
- push on the dictionaries for the object being sent the message
- execute the message, optionally returning results
- pop off the object's dictionaries
- re-install the prior object's dictionaries if there was a pending send.

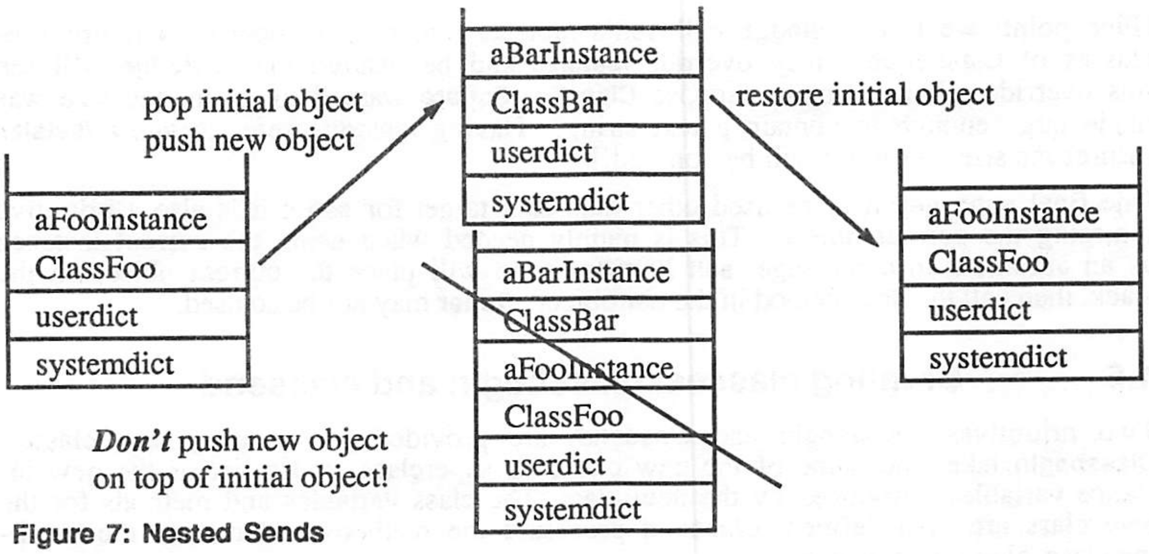


Figure 7: Nested Sends

1.5 Self and Super

Within the context of a method, Smalltalk-80 uses two pseudo variables, **self** and **super**, which may be used as targets for the **send** primitive.

The **self** pseudo variable refers to the current object. Use of **self** within a method directs send to restart name resolution at the object that caused the method to be invoked.

The **super** pseudo variable refers to superclasses of the current method's class. Use of the **super** pseudo variable tells **send** to begin looking for the method in the immediate superclass of this class.

In the following example, **ClassFoo**'s **/zot** method sends the **/foo** message to **self**. This instructs send to begin the name search for **/foo** over again at the top of the dictstack. This resolves to **ClassFooBar**'s **/foo** method. This in turn sends the **/foo** method to **super**. This instructs **send** to start the search for the **/foo** method not at the top of the dictstack, but at the immediately lower class **ClassFoo**. **Super**, therefore avoids cyclic references; and **self** allows a class to refer to its subclasses not yet defined

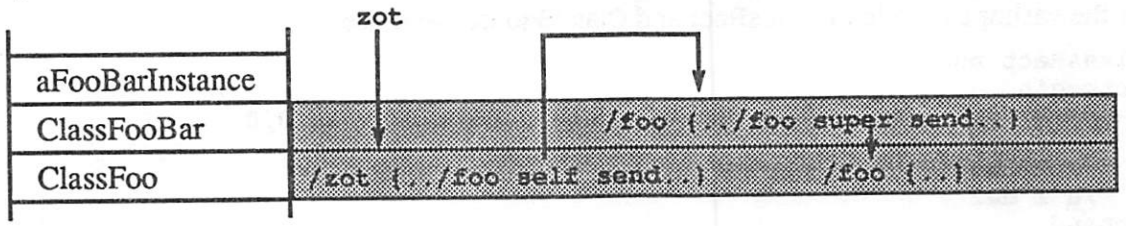


Figure 8: Self and Super Name Resolving

We'll use **self** and **super** to make two changes in the earlier **ClassSquare** example. First, we did not override **/setsize** to enforce both the height and width to be the same. This is corrected by overriding **/setsize**. Next, we change **/setedge** to call **/setsize** rather than repeating the code already in ClassRect.

```
ClassSquare:
    /SuperClasses [ClassRect]
    /setsize      {2 copy ne {<error>} {/setsize super send} ifelse}
    /setedge      {dup /setsize self send}
    /getedge      {H}
```

[Fine point: we have **/setedge** call **self**'s **/setsize** rather than **super**'s so future subclasses of **ClassSquare** may override **/setsize** and be insured that **/setedge** will see this override. For example, suppose **ClassTextSquare** wanted to insure the size was made large enough to contain a text string. Having **/setedge** refer to **self**'s **/setsize** insures the size constraint will be honored!]

One final note: **self** may be used other than as a target for **send**; it is also a primitive returning the current object. This is mainly needed when using the current instance as an argument in a message: **self /foo bar send** will place the current object on the stack, then call the **/foo** method in the **bar** object. **Super** may not be so used.

1.6 Creating classes: classbegin and classend

Two primitives, **classbegin** and **classend**, are provided for creating new classes. **Classbegin** takes the name of the new class, its superclass, and a list of the new instance variables introduced by the new class. The class variables and methods for the new class are then defined. **Classend** processes the methods and returns the resulting class, along with its name.

The list of class variables may be given either as an array or as a dictionary with initial default definitions. In the array case, the variables are initialized to **null**. For a simple class with no superclass, **null** may be used for the superclass to **classbegin**.

The use is typically:

```
/NewClass SuperClass
dictbegin
    /InsVar1 val1 def
        ...
    /InsVarN valN def
dictend
classbegin
    /ClassVar1 val1 def
        ...
    /ClassVarN valN def
    /method1 {..} def
        ...
    /methodN {..} def
classend def
```

For the earlier example of **ClassRect** and **ClassSquare** we'd use:

```
/ClassRect null
dictbegin
    /X 0 def      % initial defaults are a unit rect at 0,0
    /Y 0 def
    /W 1 def
    /H 1 def
dictend
classbegin
    /FillColor  1 0 0 rgbcolor def
    /getsize    {W H} def
    /setsize    {/H exch def /W exch def} def
    /getloc     {X Y} def
    /setloc     {/Y exch def /X exch def} def
    /getfill    {FillColor} def
    /setfill    {/FillColor exch def} def
    /paint      {X Y W H rectpath FillColor setcolor fill} def
classend def
```

```
/ClassSquare ClassRect [] % no new instance variables
classbegin
    /setsize
        {2 copy ne {pop pop (!) print}{/setsize super send} ifelse} def
    /setedge      {dup /setsize self send} def
    /getedge      {H} def
classend def
```

The processing by **classbegin** includes:

- Create the dictionary for the class
- Save the class name
- Create the SuperClasses array for the class
- Create a dictionary of instance variables that includes all the instance variables of this class and its superclasses.

Between the **classbegin .. classend** pair, the dictionary stack is set up to have all the superclasses for the class on the stack, along with the new dictionary for the class itself.

The processing by **classend** includes:

- Method compile all the class methods
- Check the **UserProfile** dictionary for overrides.
- Return the class name and dictionary

Method compilation processes each method to replace **self send** and **super send** with the appropriate PostScript fragments. In particular, **/foo self send** is converted to simply **foo**. This is because sending a message to the current object simply means executing the message with no change in the dictstack; the object is already in place!

UserProfile is a dictionary in which the user may place procedures of the same name as classes which they want to modify. The procedure is passed the class name and class dictionary to modify as desired. Thus, if we were to desire the default color for rectangles to be blue instead of red, this would be placed in the **UserProfile:**

```
        /ClassRect {dup /FillColor 0 0 1 rgbcolor put} def
```

We'll return to this in the defaults discussion below.

1.7 Creating instances: /new

New instances of classes are created by sending the message **/new** to the desired class. Thus to create a new rectangle one would use: **/new ClassRect send.** The new object is typically immediately defined as a name:

```
        /aRectInstance /new ClassRect send def
```

But we didn't define a **/new** method for **ClassRect,** so how does **/new** work? The answer is simple; we define a root class, class **Object,** which is used by all our other classes; and which has the **/new** method defined in it.

Class **Object** defines **/new** to:

- Allocate a dictionary object
- Install the initial instance variables
- Install the SuperClasses array for the instance
- Perform other initialization

Class **Object** factors these tasks into two other methods: **/newobject** performs all but the final initialization step which is performed by **/newinit**. Subclassers override **/newinit** to perform initialization specific to their class. Typically **/newinit** is used to process arguments used by **/new** for the given class. Suppose, for example, that **ClassFoo** needed initialization for its **/Foo** instance variable. It would use **/newinit:**

```
/newinit { % foo => -
    /newinit super send
    /Foo exch def
def
```

The earlier definition for **ClassRect** must be changed to include class **Object:**

```
/ClassRect Object
dictbegin
    /X 0 def
    ...
```

ClassRect needs no **/newinit** because it has no special initialization other than that gotten for free by defaulting the instance variables to a unit rectangle at the origin.

The instance variables **X, Y, W, H,** the class variable **Color,** and the methods **getsize, setsize,** .. are laid out as follows in their dictionaries for an instance of **ClassSquare:**

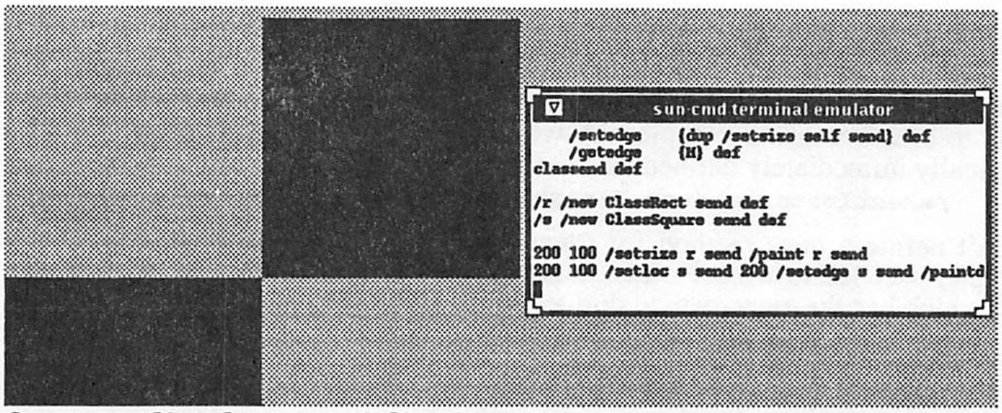
aSquare	**X Y W H**
ClassSquare	**setsize setedge getedge**
ClassRect	**FillColor getsize setsize getloc setloc getfill setfill paint**
Object	**new newobject newinit ...**

Figure 9: Name Layout in a Square Instance

To make and use a square:

```
/r /new ClassRect send def
/s /new ClassSquare send def
200 100 /setsize r send /paint r send
200 100 /setloc s send 200 /setedge s send /paint s send
```

The resulting screen looks like:



```
    /setedge    {dup /setsize self send} def
    /getedge    {H} def
classend def

/r /new ClassRect send def
/s /new ClassSquare send def

200 100 /setsize r send /paint r send
200 100 /setloc s send 200 /setedge s send /paintd
```

Screen 1: ClassSquare and ClassRect Example

1.8 Class Basics Review

We've now covered the fundamentals of NeWS classes:

- The PostScript dictionary stack provides a simple name overriding technique.
- This forms the basis for defining objects as an ordered set of dictionaries composed of the superclass dictionaries and an optional instance dictionary.
- The **send** primitive is used to automate the dictionary stack manipulation and method execution. A message has the form:
  ```
  arg1 .. argN /method object send => results
  ```
- The pseudo variables **self** and **super** are special targets for send which refer to the current object and the superclass for the current method. **Self** may also be used as a primitive returning the current object.
- **Classbegin** and **classend** are primitives used for constructing classes. The method compilation step in **classend** resolves **self** and **super** sends.
- **/new** and **/newinit** are means for constructing new instances of classes. They are provided in a special root class, class **Object**.

Most of this was present in the NeWS 1.0 release with the Lite toolkit as presented in the earlier paper on NeWS classes [3]. [There the entire class implementation was given as a two page Appendix A!] We'll now go on to further class topics brought about by the increased use of classes in TNT.

2.0 Advanced Class Topics

The NeWS Toolkit makes far greater use of classes than its Lite precursor. [One joke has it that TNT should have been called Classic!]

The flavor of TNT is that most NeWS objects (canvases, processes, events) have a class based counterpart. In particular, all user interface objects (windows, menus, buttons, ...) are based on a **ClassCanvas** subclass. In addition, the notion of a "container hierarchy" is supported via a **ClassBag** which introduces the notion of nesting of **ClassCanvas** instances. Both support event management through the notion of activation/deactivation.

X11/NeWS itself makes far greater use of classes, even implementing the X Window Manager in subclasses of **ClassCanvas**!

To support this, several enhancements were added to the basic class primitives, and to the base class, class **Object**. The following section will enumerate these changes, subsequent sections enlarging on the purpose and use of these changes.

In the following examples, we will be executing small code fragments using the NeWS P-Shell (psh) which simply pipes its **stdin** to the NeWS server, and pipes the output from NeWS to its **stdout**.

2.1 X11/NeWS Enhancements to PostScript Classes

These changes are of two kinds: changes to the class primitives and changes to the base class, class **Object**.

The class primitives are the class creation primitives **classbegin** and **classend**, the method compiler, **send**, and miscellaneous utilities in the class implementation file class.ps.

The first major change is that **classbegin** can take an array of superclasses: supporting multiple inheritance. Other changes to classbegin are:

- Dynamic reinstallation of classes
- More flexible instance variable redefinition and protection
- Support for enumeration of classes
- SuperClasses pushed onto dictstack between **classbegin/classend**

The first major change to **classend** was to migrate much of its code up to **classbegin**, thereby allowing us to push the SuperClasses onto the dictstack between **classbegin/classend**. The other major change was to allow users to modify the newly created class via the **UserProfile** dictionary.

The method compiler was rewritten to be easily enhanced; it simply keeps a dictionary of tokens it will modify, along with the code to do the modification. One such enhancement is to make sure **self send** and **super send** work correctly when local dictionaries have been pushed onto the dictstack.

The **send** primitive was rewritten in C for performance reasons. Access to the send stack was provided. Nested sends were popped off the dictstack.

The changes to class Object include:

- **/installmethod** for creating on-the-fly methods
- **/new**: factored into **/newobject, /newinit**
- **/newmagic**: method for creating instances from existing dictionaries
- **/new**: now copies compound instance variables
- **/newdefault, /defaultclass** methods for use with abstract classes
- **/sendtopmost**: access to the send history stack
- **promote, promoted? unpromote ?promote:** utilities for promoting class variables to instance variables
- **/destroy** & **/obsolete:** methods for object cleanup.

2.2 Classbegin: Multiple Inheritance

Classbegin was changed to allow the superclass to be an array of classes. The array may be empty, indicating a root class. The resulting class is in no way special; it simply has a SuperClasses array that contains more class dictionaries than if it was a subclass of a single superclass. Thus, for NeWS, multiple inheritance conceptually differs little from simple inheritance. In fact, the initial rewrite for multiple inheritance was smaller than the single inheritance version due to fewer special cases!

The rules for the resulting SuperClasses list given the initial superclass array are:

- If ClassA precedes ClassB in **classbegin**'s superclass array, then it will proceed ClassB in the resulting SuperClasses list.
- If ClassA precedes ClassB in any of the SuperClasses lists for a class in **classbegin**'s superclass array, then it will proceed ClassB in the resulting SuperClasses list.

Two fundamental changes to **classbegin** and **classend** were required:

- The incoming array had to be converted to a SuperClasses list using the above rules
- A more sophisticated reduction of **super send** by the method compiler was needed.

The primary use of multiple inheritance is to mix-in a common capability into a set of classes that do not descend from each other. As an example, consider the following class whose only purpose is to add a text string to graphical classes:

```
/ClassLabel Object
dictbegin
    /Label      () def
dictend
classbegin
    /LabelColor 0 0 1 rgbcolor def
    /LabelX     {X 10 add} def
    /LabelY     {Y 10 add} def

    /paint      {/paint super send /paintlabel self send} def
    /paintlabel {LabelColor setcolor LabelX LabelY moveto Label show}def
    /getlabel   {Label} def
    /setlabel   {/Label exch def} def
classend def
```

Note that **ClassLabel** is a subclass of **Object** (which has no **/paint** method), but **ClassLabel**'s **/paint** calls super! **ClassLabel** would fail if sent **/paint**, but instead must be mixed in with another class which does have a **/paint** method. Although we could simply give **ClassLabel** no superclass, we decide to descend from **Object** so that some messages will work, such as **/classname** below, even though others will not.

Now let's define a button class:

```
/ClassSimpleButton [ClassLabel ClassRect]
dictbegin
    % New button instance variables
dictend
classbegin
    % New button class variables & methods
classend def
```

To see its superclasses:

```
    /superclasses ClassSimpleButton send
    [exch {/classname exch send} forall] ==
[/Object /ClassRect /ClassLabel]
```

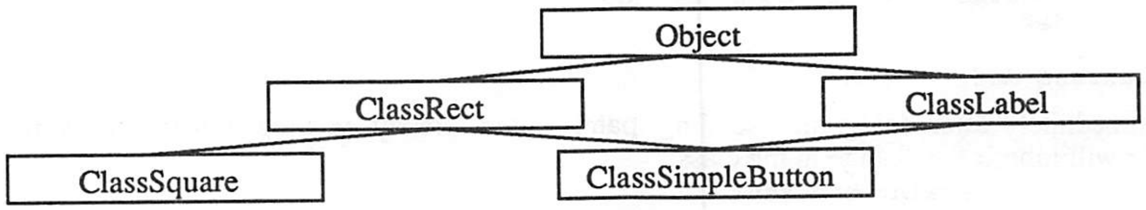
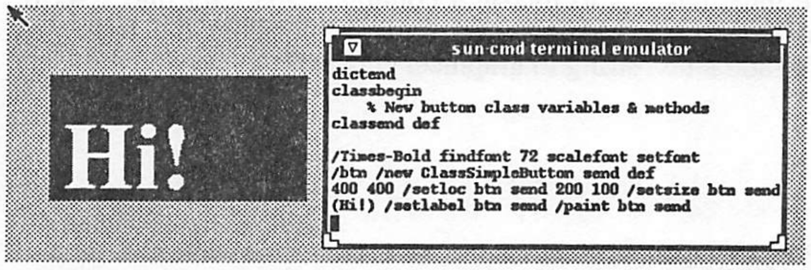


Figure 10: Multiple Inheritance

To use **ClassSimpleButton** & **ClassLabel**:

```
    /Times-Bold findfont 72 scalefont setfont
    /btn /new ClassSimpleButton send def
    400 400 /setloc btn send 200 100 /setsize btn send
    (Hi!) /setlabel btn send /paint btn send
```



```
dictend
classbegin
    % New button class variables & methods
classend def

/Times-Bold findfont 72 scalefont setfont
/btn /new ClassSimpleButton send def
400 400 /setloc btn send 200 100 /setsize btn send
(Hi!) /setlabel btn send /paint btn send
```

Screen 2: Simple Button

[Historic note: the initial idea for multiple inheritance, as well as several (at least 7!) other changes in the X11/NeWS class implementation, was from the LispScript re-implementation of class.ps done at Schlumberger by David Singer and Rafael Bracho[9]. This project, like TNT, had far greater need for sophisticated classes. Although we were driven to modify the original Schlumberger implementation, their initial implementation was invaluable!]

2.3 Classbegin: Reusable Class Dictionaries

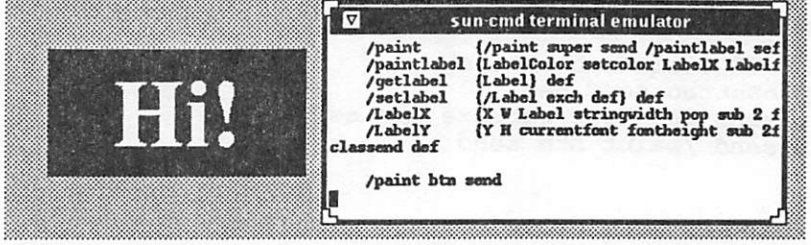
In PostScript, compound objects such as strings, arrays, and dictionaries are shared: the objects point to a piece of virtual memory. NeWS added the capability to PostScript dictionaries to grow in size. These two capabilities combine to allow classbegin to reuse a class dictionary when the class is re-defined.

This has interesting ramifications. If I rebuild a class, all its existing subclasses will see these changes. Let's rebuild the **LabelClass** defined above to center the text in its bounding box. We do this by redefining **LabelX** and **LabelY** to calculate the correct position:

```
/ClassLabel Object
        ...
    /LabelX {X W Label stringwidth pop sub 2 div add} def
    /LabelY {
        Y H currentfont fontheight sub 2 div add
        currentfont fontdescent add
    } def
    ...
classend def
```

Immediately after doing this, sending **/paint** to the already existing **/btn** in the example will inherit the change in the class:

```
            /paint btn send
```



```
    /paint       {/paint super send /paintlabel sef
    /paintlabel  {LabelColor setcolor LabelX Labelf
    /getlabel    {Label} def
    /setlabel    {/Label exch def} def
    /LabelX      {X W Label stringwidth pop sub 2 f
    /LabelY      {Y H currentfont fontheight sub 2f
classend def

    /paint btn send
```

Screen 3: Simple Button with Centered Label

2.4 Classbegin: Redefinition of Instance Variables

In earlier versions of classes, clients could inadvertently reuse an instance variable name without intending to do so. It was decided to warn clients when they introduce an instance variable whose name has already been used by a superclass.

Unfortunately there was good reason for knowledgeable clients to redefine instance variables as an efficient way for their particular class to have a different initial value for that variable. A good example would be for **ClassRect** and **ClassSquare**; especially if **ClassRect** had decided to use non-square values for the height and width! Another example would be to allow **ClassSimpleButton** to have its label initially be (Button!):

```
/ClassSimpleButton [ClassLabel ClassRect]
dictbegin
    /Label (Button!) redef
dictend
classbegin
    % New button class variables & methods
classend def
```

To solve this problem, a new primitive was introduced: **redef**. When instance variables are given to **classbegin** in the dictionary form, use of **redef** will allow the variable to be redefined with no warning; and without changing the initial value of the superclass's instance variable. Using **redef** on a new name results in a warning.

Fine point: a redefined instance variable is redefined *in the new class only*, not in the class that initially defined the variable. Thus, in the above, **ClassLabel** has its **Label** still initialized to the empty string, while **ClassSimpleButton**, and its descendents, will have **Label** initialized to (Button!):

```
    /new ClassLabel send /getlabel exch send ==
()
    /new ClassSimpleButton send /getlabel exch send ==
(Button!)
```

In sum, then, we issue warnings when a name is reused as an instance variable; use of **redef** allows redefinition of an existing instance variable's initial value; and a warning will be issued when **redef** is used with a name that does not already exist.

2.5 Classbegin: SubClasses list

In NeWS 1.0, each class kept a list of the names of its subclasses. This was to allow enumeration of the class tree, and to support browsers. Class names, rather than direct references to the class objects, were used to avoid garbage collection problems. Because X11/NeWS has features to solve the garbage collection problems, **classbegin** now installs references to the objects, not simply their names.

Two utility methods of class **Object** may be used to enumerate the class hierarchy: **/superclasses** and **/subclasses**, both of which return arrays of objects.

To print out the names of the superclasses of **ClassMenu**, for example:

```
    /superclasses ClassMenu send
    [exch {/classname exch send} forall] ==
[/Object /ClassTarget /ClassCanvas /ClassSelList]
```

and to print out the names of the subclasses of **ClassCanvas**:

```
    /subclasses ClassCanvas send
    [exch {/classname exch send} forall] ==
[/ClassFramebuffer /ClassSelList /ClassControl /ClassBag /ClassXWindow]
```

2.6 Classend: UserProfile

As part of the initialization of X11/NeWS, a user provided file .startup.ps is read. This file may add entries to the **UserProfile** dictionary created just prior to reading the file. **Classend** looks to see if there is a procedure of the same name as the class. If so, it hands the class to the procedure for modification.

If I preferred my rectangles to start out green, I would place this in .startup.ps:

```
UserProfile begin
    /ClassRect { % class => class'
        0 1 0 rgbcolor /setfill 2 index send
    } def
end
```

This is part of a general defaulting scheme discussed below.

2.7 Object: /newinit

In NeWS 1.0, every class wanting to initialize new instances by executing code in the **/new** call had to override **/new** in a standard cliche:

```
/new { % <args> => instance
    /new super send begin
        <instance initialization code>
        currentdict
    end
} def
```

This, though clever, began to be very unwieldy for complex argument handling. It also was difficult to explain to new programmers: the **/new super send begin** simply happened to fake a **send** to the new instance. Lastly, it became a performance problem for classes that had many superclasses.

The solution was to recognize that **/new** really does two separate things: it sends a message to a class to create an empty instance of that class, then it sends a second message to the newly created instance to initialize itself. We achieve this by factoring **/new** into two methods: **/newobject** to do the former, and **/newinit** to do the latter:

```
/new { % <args> => instance
    /newobject self send                % <args> instance
    {/newinit self send self} exch send % instance
} def
```

The default **/newobject** in class **Object** creates a new instance dictionary, copies the instance variables into that dictionary, and sets the SuperClasses array. **/newinit** defaults to a no-op; expecting superclasses to override it. The second line in **/new** is a clever way to send **/newinit** to the new object, and to return the new object without keeping it on the operand stack.

2.8 Object: /newmagic

NeWS extensions to PostScript are done as "magic" dictionaries similar to PostScript font dictionaries. The three primary extensions are canvases, events, and processes. Not surprisingly, there is a corresponding class object for each of these.

In NeWS 1.0 these classes had an instance variable corresponding to the NeWS magic dictionary. Thus **ClassCanvas** required a **/Canvas** instance variable.

The factoring of **/new** into **/newobject** and **/newinit**, and X11/NeWS allowing magic dicts to be extended by users (not allowed in NeWS 1.0) combine to make possible converting an existing magic dict (canvas, event, process) into an instance of an object. This is formalized by the method **/newmagic** in Object.

The **/newmagic** method converts a dictionary argument into an un-initialized instance of the class to which the message is sent. Thus ClassCanvas creates a canvas using the NeWS primitive newcanvas, then sends **/newmagic** to self to convert it into an instance of the subclass of **ClassCanvas** being created. This is done by having **ClassCanvas** supply an override for **/newobject.** Similarly, **ClassInterest** and **ClassEventMgr** convert events and processes into class instances.

To see how it works, we'll create a simple dict with one field. We'll then create a new **ClassRect** object from that dict. The resulting dict is the same as the initial dict but enhanced to be an instance of **ClassRect**.

```
    /d 10 dict def
    d /Magic (Magic) put
    [d {pop} forall] ==
[/Magic]
    /dd d /newmagic ClassRect send def
    [dd {pop} forall] ==
[/Magic /X /Y /W /H /SuperClasses]
    d dd eq ==
true
```

2.9 Object: Destruction and Obsolescence

Four new methods have been added to class **Object** to assist in managing object destruction. To better understand these, we need to review NeWS memory management.

PostScript objects are either simple or compound. Simple objects (ints, reals, bools, ..) require no additional storage, while compound objects (strings, arrays, and dictionaries) are collections of other objects and reference additional memory in PostScript's virtual memory. When compound objects are replicated by being pushed onto a stack or put into a dictionary or array, only the object, not the associated virtual memory, is replicated. This virtual memory segment contains a reference count equal to the number of references to it. When this count becomes zero, the memory is released.

Two difficulty with reference counted garbage collection arise:

- Cycles: Memory cycles may easily occur in which one compound object contains a reference to another object which in turn refers back to it. If the first reference to the object is removed, the cyclical reference prevents garbage collection.
- Managers: Other external agencies outside the applications control may need to reference a compound object. Examples are selection and keyboard focus managers. The reference they maintain can cause an object to not be garbage collected.

X11/NeWS introduces the notion of "soft" references: when the references to an object are all soft, a special **/Obsolete** event is generated. A process may listen for these, performing an appropriate action.

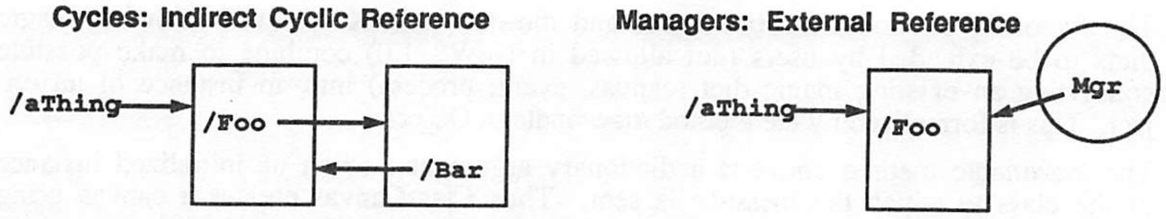

Cycles: Indirect Cyclic Reference
/aThing
/Foo
/Bar
Managers: External Reference
/aThing
/Foo
Mgr

Figure 11: Cycles and Managers

To help manage instance destruction, four methods have been added to class **Object**:

- **/destroy**: Called when an object is to be no longer used. Typically this will kill processes spawned by the object, remove themselves from managers referencing them, and break cyclic references. Defaults to a no-op in **Object**.
- **/destroydependent**: When **/destroy** is sent to an instance containing references to other instances, **/destroydependent** rather than **/destroy** is sent to these other instances. If an instance contains a menu, for example, sending the menu **/destroydependent** allows the menu to not destroy itself if it is shared among other instances. This allows all **OpenLookFrames**, for example, to use a single menu. Defaults to **destroy self send in Object**.
- **/obsolete**: Sent to an object when all remaining references are soft. Defaults to **destroy self send in Object**.
- **/cleanoutclass**: Prepare a class dictionary for re-use. Typically this simply calls **undef** for each key in the class dictionary.

Finally, there is a **classdestroy** primitive added to the **classbegin classend** pair. It removes the class from any SubClasses lists that contain it, then calls **/cleanoutclass**.

2.10 Object: /installmethod

Because **UserProfile** modifications of classes required installing new methods into existing classes, and because certain instances need to override their class's methods, we decided to make method compilation generally available through a method in class **Object**.

The **/installmethod** method takes a procedure as its argument and sends it to the class for which the procedure should be methodcompiled. This resolves **self send** and **super send** correctly for that class. The resulting method is then defined in that class.

Although **/installmethod** is generally sent to a class, we also allow it to be sent to instances of a class. These "instance methods" resolve **super** to be the class of the instance, not the superclass of the instance's class. Allowing instance methods is a pragmatic way to avoid almost-empty classes that override only one method.

As an example, suppose we decided not to create a separate class for squares, but I, as a client of **ClassRect**, had to insure a particular rectangle was square. I'd do this as follows:

```
/square /new ClassRect send def
/setsize {
    2 copy ne {
        pop pop (Please make me square.\n) print
    } {
        /setsize super send
    } ifelse
} /installmethod square send
```

Resulting in this behavior:

```
    10 100 /setsize square send
Please make me square.
    45 45 /setsize square send
    [/getsize square send] ==
[45 45]
```

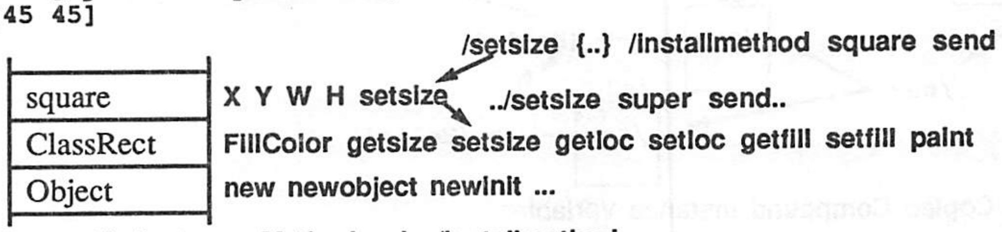

/setsize {..} /installmethod square send
square
ClassRect
Object
X Y W H setsize
../setsize super send..
FillColor getsize setsize getloc setloc getfill setfill paint
new newobject newinit ...

Figure 12: Instance Methods via /installmethod

2.11 Object: /new Copies Compound Instances

Recall that PostScript compound objects are shared; they return pointers to dynamically allocated memory. In NeWS 1.0, when a pre-initialized instance variable was a compound object, each new instance shared this object. In X11/NeWS, /new makes separate copies of these objects. This avoids having one instance modifying a compound object being propagated to all instances. The correct way to achieve this shared behavior is to use class variables.

To see this in action, notice that modifying one of two strings referring to the same data changes both strings:

```
    /s1 (Hello) def
    /s2 s1 def
    [s1 s2]==
[(Hello) (Hello)]
    s1 0 104 put % 104 = 'h'
    [s1 s2]==
[(hello) (hello)]
```

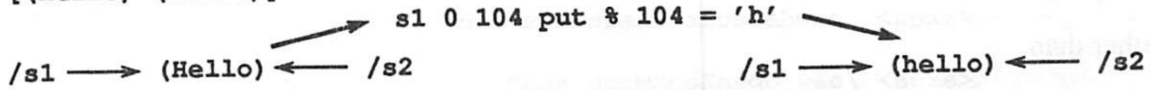


Figure 13: Shared Strings

Now make the beginnings of a string class with a string (compound) instance variable:

```
/SimpleString Object
dictbegin
    /Str (Hello!) def
dictend
classbegin
    /getstring {Str} def
    % .. & more stuff
classend def
```

The analogous tests with string instances will not modify both strings because of the copying of compound instance variables:

```
    /s1 /new SimpleString send def
    /s2 /new SimpleString send def
    [/getstring s1 send /getstring s2 send] ==
[(Hello!) (Hello!)]
    /getstring s1 send 0 104 put % 104 = 'h'
    [/getstring s1 send /getstring s2 send] ==
[(hello!) (Hello!)]
```

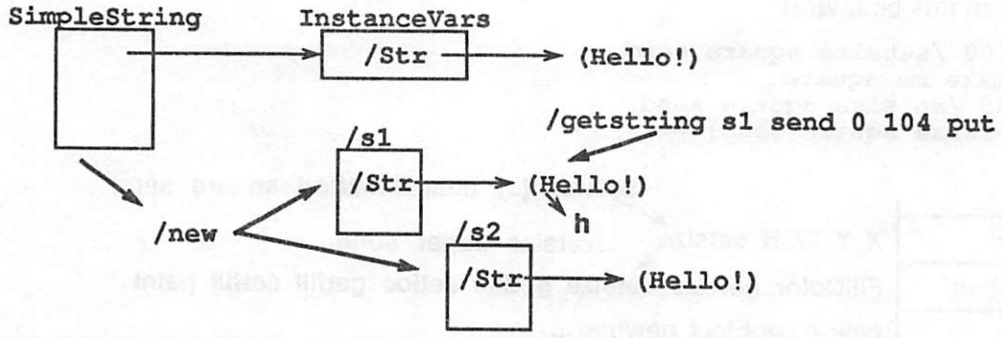


Figure 14: Copied Compound Instance Variables

2.12 Object: /newdefault & /defaultclass

The Smalltalk-80 notion of abstract superclasses is supported in NeWS. An abstract superclass is simply a class that requires one or more methods to be defined, but does not implement them; expecting a subclass to do so. When such a method is not implemented by subclasses, the **SubClassResponsibility** error is raised. An abstract superclass is therefore incomplete and thus should never receive the **/new** method.

It was decided to complement this notion with a special method, **/newdefault**, that would redirect the **/new** message to the correct subclass of the abstract superclass. It is implemented by having a **/DefaultClass** class variable for each class. This is initially **self** for all classes unless overridden. Abstract superclasses always override **/DefaultClass**.

This is used to provide Look & Feel (user interface) independence in TNT. All UI elements have an abstract superclass. **ClassMenu** is such an abstract class with **OpenLookMenu** being its default implementation. Thus clients are encouraged to create new menus via:

```
        <args> /newdefault ClassMenu send
```

rather than

```
        <args> /new OpenLookMenu send
```

One oddity of **/DefaultClass** is that the class cannot be defined until the subclass has been defined. This is solved by setting **/DefaultClass** to be a procedure returning the desired subclass, deferring looking for the subclass until after the abstract class is defined. Thus, in **ClassMenu** we find:

```
        /DefaultClass {OpenLookMenu} def
```

There are times when a client wants to make a subclass of one of these default classes. This is done using the **/defaultclass** method. Thus, if I needed to modify **ClassMenu**'s default class, I'd use:

```
        /MyMenu /defaultclass ClassMenu send
        dictbegin
        ...
```

This is a generalization of a concept that started in NeWS 1.0 as **DefaultWindow** and **DefaultMenu** which were simply pointers to the desired default classes.

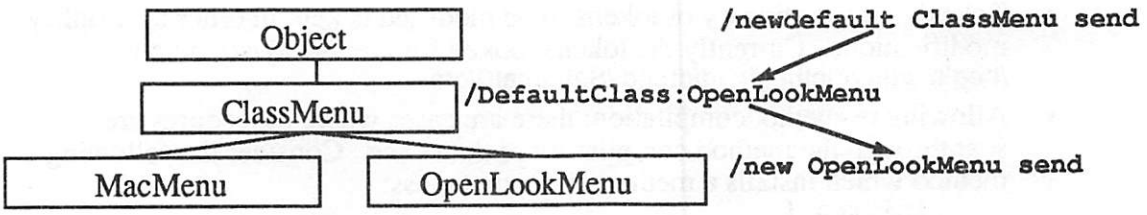


Figure 15: /newdefault and /defaultclass

2.13 Object: Promotion of Class Variables

Because instances are simply dictionaries, a class variable may be overridden by defining the same name in the instance dictionary. Consider rectangle class and instance variables:

`aRectInstance`	`X Y W H`	Instance Variables
`ClassRect`	`FillColor`	Class Variables

Now assume we **def** into **aRectInstance** the name **FillColor.** This is called *promoting* the ClassVariable **FillColor** to be an Instance Variable. Its value will override the class variable whenever referenced in methods for that one instance.

Class **Object** provides four utilities for handling promotion:

- **promote**: promote a class variable to an instance variable.
- **promoted?**: check if the variable is an instance variable.
- **?promote**: promote variable if it differs from the class version.
- **unpromote**: remove variable from instance variables.

We'll see how TNT uses these in the discussion of defaults below.

2.14 send: Change Summary

The discussion on send in the first section presented the X11/NeWS behavior. Changes since NeWS 1.0 include:

- Nested sends: NeWS 1.0 did not pop off the current send; it simply pushed the object on top of the current object.
- Access to the popped off sends is provided. This is useful when an instance wants to know what object caused it to be sent a message. [This is a generalization of the **ThisWindow** hack of NeWS 1.0]
- C implementation: For performance reasons, the **send** utility, and the support for **super send** is implemented in C.
- PostScript versions of the C implements are provided to allow overriding the C implementations. This is useful for testing and debugging.

2.15 methodcompile: Change Summary

The **methodcompile** primitive in class.ps is used by both **classend** and class **Object**'s **/installmethod** discussed above. The method compiler is almost trivial: it simply scans an array looking for **self send** and **super send**, converting them into the appropriate code fragments. Changes since NeWS 1.0 include:

- Tokenized: a dictionary of tokens to be modified is kept in order to simplify modifications. Currently the tokens looked for are: **/send /supersend /begin /end /dictbegin /dictend /SetLocalDicts**
- Allowing re-methodcompilation: there are cases where procedures are sent through the method compiler more than once. Consider the following method which installs a method in another class:

  ```
      /initfoo {
          /mumble {...} /installmethod Foo send
      } def
  ```

  The code {...} will be multiply method compiled: once during the method compilation of **/initfoo** at the **classend** of its class definition, and once when the **/installmethod** is executed.
- Local Variables: The method compiler now notices when dictionaries are added to the dictstack within a method; generating the correct code for **self** and **super** sends for this situation. It also provides a compiler directive to override the count of these local dicts.

The local variable dictionary problem is subtle: when a method needs to define local variables, it has to add a dictionary to the dictstack. Suppose **ClassSquare** had:

```
/foo {
    10 dict begin
        /a .. def
            ...
        a /setedge self send
            ...
    end
} def
```

This results in a dictstack of:

```
<10 dict>
aSquare
ClassSquare
ClassRect
...
```

But the method compiler would usually convert the **a /setedge self send** to **a setedge**. Alas, the setedge method will cause **def** to be used to set **W** and **H** ..but now in the wrong dict: the temporary local variable dict! The solution is to count the **begin/end** pairs (and **dictbegin/dictend** pairs) to determine whether the conversion of **/foo self send** to **foo** is valid. Clearly this is heuristic at best, so there is an explicit override available: **/SetLocalDicts** is used as a directive to the method compiler to override the **begin/end** counter.

3.0 Application of Classes

In the two previous sections we have seen the basic classing system in NeWS, and have looked at advanced topics brought about by increased use of classes in X11/NeWS and The NeWS Toolkit. The rest of the paper will show how these are applied.

3.1 Defaults

Unix is a very adaptable environment. Users expect to be able to modify the behavior of their keyboard (stty) and terminal emulations (termcap) easily, and to specify default behavior of programs with "*.rc" files.

Unix window systems have tried to provide flexibility through use of files containing default settings for constant parameters such as text fonts, text sizes, and colors for foreground, background and text. Difficulties arise, however, when trying to specify changing just the text size of menus, or the background color for only the text editor. Typically every program has to anticipate each parameter the user may want to override, and check some sort of defaults database for an entry. These systems tend to be limited by their database implementation, have serious performance problems, and generally have a bad fit to the underlying window semantics.

Classes provide a simple formalism to easily solve these problems. Defaults are simply class variables. Subclasses may override these variables to yield a hierarchy, differentiating Button fonts from TextEditor fonts. Individual instances may override a class variable by promoting it to be an instance variable. Users may override a class's choice of an instance variable by use of a **UserProfile** entry. Default implementations in abstract superclasses provide a formalism for user-interface intrinsics.

Simple as this appears, it is an entirely general formalism for the defaults problem! The model is this:

- Each object in the system has an associated class.
  Example: UI objects are based on **ClassCanvas** which contains general drawing parameters: **FillColor, StrokeColor, TextFont, TextColor,** etc.
- These objects are subclassed to form more differentiated objects.
  Example: **ClassMenu** is based on **ClassCanvas.**
- Each new application defined in the system defines a basic class for itself shared among all instances of that application.
  Example: **TextEditor** is a subclass of **ClassCanvas.** It contains a **TextEditorMenu** menu that is a subclass of **ClassMenu.**
- Users override these general facilities as their classes are defined using the **UserProfile.**
  Example: I can change general **FillColors** to 50% gray, the menu **FillColor** to white, and the **TextEditMenu** to red by placing in .startup.ps:

```
UserProfile begin
    /ClassCanvas { % name class => name class
        dup /FillColor .5 .5 .5 rgbcolor put
    def
    /ClassMenu { % name class => name class
        dup /FillColor 1 1 1 rgbcolor put
    } def
    /TextEditMenu { % name class => name class
        dup /FillColor 1 0 0 rgbcolor put
    } def
end
```

- Finally, individual instances can have their own, individual, values of the default class variable by promoting that variable to be an instance variable.
  Example: To color the currently selected window's clients yellow, execute this via a psh:

```
/selectedframes ClassFrame send {
    client exch send
    /FillColor 1 1 0 rgbcolor /promote 3 index send
    /damage exch send
} forall
```

  Note: This typically would be done by supplying the invocation of the application with an argc/argv argument. There is not currently a general argument passing technique in TNT that would automate this. On the other hand, use of the selected frames is a reasonable UI style.

A more subtle form of defaulting is changing the default implementation of an abstract super class. These classes typically act as so called "intrinsics", classes that define basic behavior but with the details to be filled in by specific implementations. Again, we use the class hierarchy to manage intrinsics and user control over them:

- Assume there are two styles of window frames available: **MacFrame**'s and **OpenLookFrame**'s The user chooses between them by:
```
UserProfile begin
        /ClassFrame { % name class => name class
            dup /DefaultClass {MacFrame} put
        def
end
```
  The application insures this default is used by:
```
        /win ... /newdefault ClassFrame send def
```

3.2 Magic Dict Instances

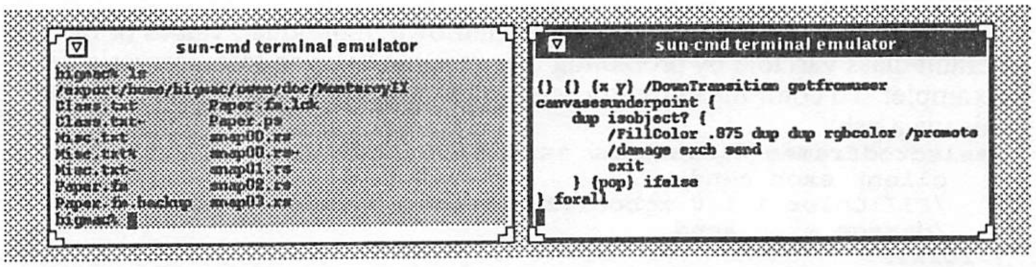
Through use of **/newmagic**, classes may choose to package instances in the dictionaries NeWS itself uses for the corresponding objects. The three most pervasive examples of this in TNT are:

- **ClassCanvas** is the common ancestor class for all UI objects. It overrides **/newobject** to use **newcanvas** to create a NeWS canvas object, turning this into the instance dictionary using **/newmagic.**
- **ClassEventMgr** forks a process which expresses interest in receiving certain events. The resulting process object is converted into the instance dictionary with **/newmagic.**
- **ClassInterest** uses the NeWS **createevent** primitive and **/newmagic** to return an instance that is both an object and a NeWS interest.

This style allows a single object to be a simple NeWS primitive and also to receive messages, and conversely, to allow a method to give **self** to a NeWS operator expecting a NeWS primitive.

As an example, the following changes the **FillColor** of the clicked-on canvas:

```
{} {} {x y} /DownTransition getfromuser
canvasesunderpoint {
    dup isobject? {
        /FillColor 0 1 0 rgbcolor /promote 3 index send
        /damage exch send
        exit
    } {pop} ifelse
} forall
```



```
bigmac% ls
/export/home/bigmac/owen/doc/MontereyII
Class.txt        Paper.fm.lck
Class.txt~       Paper.ps
Misc.txt         snap00.rs
Misc.txt%        snap00.rs~
Misc.txt~        snap01.rs
Paper.fm         snap02.rs
Paper.fm.backup  snap03.rs
bigmac%
```

```
{} {} {x y} /DownTransition getfromuser
canvasesunderpoint {
    dup isobject? {
        /FillColor .875 dup dup rgbcolor /promote
        /damage exch send
        exit
    } {pop} ifelse
} forall
```

Screen 4: Coloring Magic Dict Canvases

It works by:

- Calling the TNT **getfromuser** utility to get a mouse click, returning the x,y location.
- Calling the NeWS **canvasesunderpoint** utility to get an array of the canvases under that point.
- Using **forall** to enumerate this list looking for the first canvas that is also an object (instance of a subclass of **ClassCanvas**) via the **isobject?** utility.
- Using this canvas as an object, promoting its **FillColor** to be green, then damaging it to cause it to repaint.

3.3 Inheriting Through The Container Hierarchy

As mentioned above, all user-interface objects in TNT are subclasses of **ClassCanvas.**

Because **ClassCanvas** and its subclasses are created from NeWS canvas magic dicts, the NeWS canvas fields automatically become instance variables for instances of these classes. Several of these magic instance variables may be used to enumerate the canvas tree: **ParentCanvas, BottomCanvas, TopCanvas, CanvasAbove, CanvasBelow,** and **TopChild.**

The **ParentCanvas** field points to the canvas's immediately containing canvas in the canvas tree. We use this to form an alternative form of inheritance: inheritance through the container hierarchy rather than inheritance through the class hierarchy. Two examples of this are colors and fonts: **ClassCanvas** subclasses inherit their initial font, both family and point size, by enquiring their ParentCanvas:

```
/TextFamily{ % - => name
    /TextFamily Parent send
} def
```

Notice that the buck has to stop somewhere! The solution is to define a **ClassFrameBuffer** that is the root of the canvas tree, use **/newmagic** to bind its instance to the top of the container hierarchy, and to initialize its **TextFamily** to a standard font. Thus the framebuffer is to the container hierarchy what class Object is to the class hierarchy: the root of an inheritance tree.

3.4 Container Class: ClassBag

ClassBag provides these facilities:

- It provides named access to its children
- It provides recursive descent painting and damage repair
- It provides inherited event management
- It provides primitive layout and validation
- It manages non-canvas children; subclasses of **ClassGraphic,** which is similar to the X Gadget object.

To see these things in action, let's play with canvases and bags.

First, let's make a simple instance of **ClassCanvas.** It will be the default style: a rectangule that paints itself by stroking and filling itself using the **/FillColor** and **/StrokeColor** class variables.

```
    /can framebuffer /new ClassCanvas send def
    50 50 100 100 /reshape can send
    /map can send
    /activate can send
```

Why don't I see it initially? It's been activated, thus should paint itself when it receives damage when initially mapped. But it needs to be made opaque to enable its receiving **/Damaged** events causing it to repaint itself:

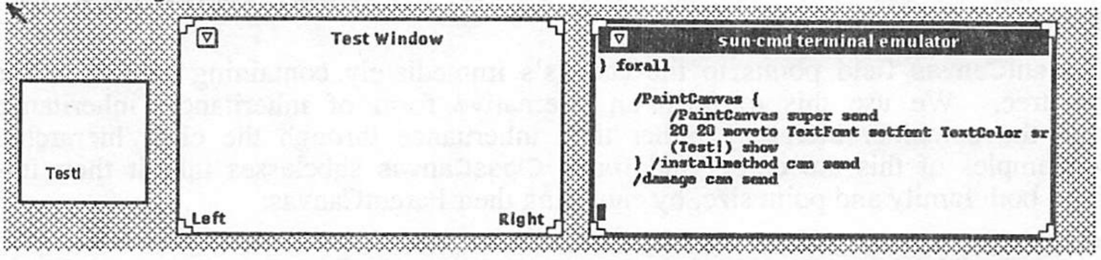
```
    false /settransparent can send
```

Now move some windows over it and watch it repair itself. In fact, let's make our own window, with no client (application) installed yet:

```
    /win null [] framebuffer /newdefault ClassBaseFrame send def
    200 200 300 200 /reshape win send
    (Test Window) /setlabel win send
    (Left) (Right) /setfooter win send
    /activate win send
    /map win send
```

Move **win** on top of **can** to see its damage repair work. Play some. To personalize our canvas, let's replace its painting procedure:

```
    /PaintCanvas {
        /PaintCanvas super send
        20 20 moveto TextFont setfont TextColor setcolor
        (Test!) show
    } /installmethod can send
    /damage can send
```



```
} forall

    /PaintCanvas {
        /PaintCanvas super send
        20 20 moveto TextFont setfont TextColor sr
        (Test!) show
    } /installmethod can send
    /damage can send
```

Screen 5: A ClassCanvas and ClassBaseFrame Pair

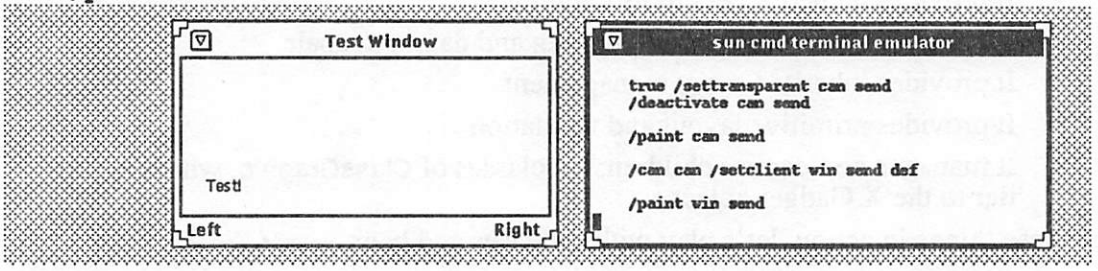
This installs the **PaintCanvas** method (called by **/paint** and the damage repair procedures) in the instance, having it call **super** to paint the stroke and fill.

OK, let's make **can** hop into **win** by making it become the client! First make **can** transparent again and deactivate it to show that it can't take care of itself anymore:

```
    true /settransparent can send
    /deactivate can send
```

Note that the canvas disappeared! This is because changing the state of the transparency of a canvas propagates damage to the framebuffer, which repaints itself, hiding the canvas. To see it:

```
    /paint can send
```



```
    true /settransparent can send
    /deactivate can send

    /paint can send

    /can can /setclient win send def

    /paint win send
```

Screen 6: Reparenting the ClassCanvas into the ClassBaseFrame

Now drag **win** over it; no damage repair. We've successfully deactivated it. To make **can** the client of win, call **/setclient** which returns the old client which will be null in this case.

```
    /can can /setclient win send def
    can ==
null
```

It won't paint automatically; you'll have to ask **win** to paint itself. Note we no longer have a handle on our canvas!

```
    /paint win send
```

Now reshape **win** and move it around; **can** really did get glued in and reactivated correctly.

We can access **can** by the name **/Client**. The method **/sendclient** sends messages to named clients, allowing us to indirectly manipulate **can** through **win**:

```
    /Times-Bold 24 null /settextparams /Client /sendclient win send
    /paint win send
```

We can pop can back onto the framebuffer by:

```
    /can null /setclient win send def
    /paint win send
    framebuffer /reparent can send
    50 50 100 100 /reshape can send
    /map can send
    /paint can send
```

We've now shown all the advertised features of bag except use of **ClassGraphic**. This is left as an exercise for the reader!

3.5 pswm: TNT Class Based X Window Manager!

The pswm X window manager[7] is implemented in PostScript using the TNT toolkit! This is done by creating two sets of classes:

- **XClientWindow** & **ClassXWindow**: **XClientWindow** is a subclass of **ClassXWindow** which is in turn a subclass of **ClassCanvas**. An instance of **ClassXWindow** is created by using **/newmagic** on an existing X window. Because X windows are canvas objects like any other NeWS canvas, we can make an instance out of X window magic dicts as we would with any other NeWS canvas.
- **XFrame**: a subclass of **ClassFrame** which is specialized for X window manager functions. There are five additional subclasses which map onto the basic frame types: **XBaseFrame, XCommandFrame, XHelpFrame, XNoticeFrame**, and **XIconFrame**.

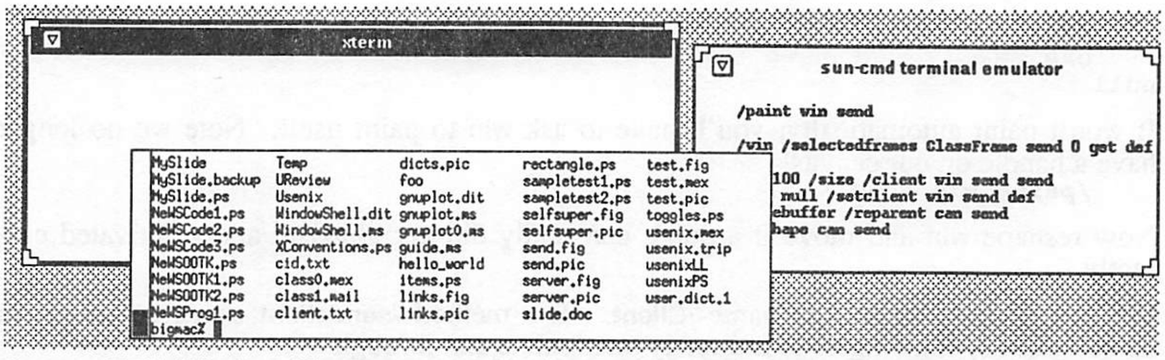
We can manipulate these clients just as we did above with simpler clients:

First, start an XTerm, select its frame, and get a reference to it. (To make it interesting, give it a scrollbar and fill it up with listings or something else.)

```
    /win /selectedframes ClassFrame send 0 get def
```

Now pop the XTerm client onto the framebuffer:

```
    100 100 /size /client win send send
    /can null /setclient win send def
    framebuffer /reparent can send
    /reshape can send
```



```
/paint win send

/win /selectedframes ClassFrame send 0 get def
100 /size /client win send send
 null /setclient win send def
ebuffer /reparent can send
hape can send
```

```
MySlide          Temp              dicts.pic        rectangle.ps     test.fig
MySlide.backup   UReview           foo              sampletest1.ps   test.mex
MySlide.ps       Usenix            gnuplot.dit      sampletest2.ps   test.pic
NeWSCode1.ps     WindowShell.dit   gnuplot.ms       selfsuper.fig    toggles.ps
NeWSCode2.ps     WindowShell.ms    gnuplot0.ms      selfsuper.pic    usenix.mex
NeWSCode3.ps     XConventions.ps   guide.ms         send.fig         usenix.trip
NeWSOOTK.ps      cid.txt           hello_world      send.pic         usenixLL
NeWSOOTK1.ps     class0.mex        items.ps         server.fig       usenixPS
NeWSOOTK2.ps     class1.mail       links.fig        server.pic       user.dict.1
NeWSProg1.ps     client.txt        links.pic        slide.doc
bigmac%
```

Screen 7: X Window Management with ClassBag and ClassCanvas

The first line is to get the location and size set up for the following **/reshape** so that it will be the same size. No repainting was required because the X clients are opaque.

Now play with the X client typing at it and scrolling. Finally put it back into its window:

```
    /can can /setclient win send def
```

3.6 Obsolete Object Management

Class **Object** maintains a process which looks for all **/Obsolete** events. It then sends **/classdestroy** to classes, or **/obsolete** to instances. It does nothing for non-class objects.

To demonstrate how obsolete management works; we'll look at how canvases get handled.

First, make a simple canvas as before, installing a message to be printed if **/obsolete** is encountered:

```
    /MakeCan {
        /can framebuffer /new ClassCanvas send def
        50 50 100 100 /reshape can send
        /map can send
        /activate can send
        false /settransparent can send
        /obsolete {
            (I'm going away!\n) print
            /obsolete super send
        } /installmethod can send
    } def
```

Now make one, then destroy it. The **/destroy** method will remove all indirect references to the object by deactivating the canvas, thus having no process point to the canvas.

```
    MakeCan
    /destroy can send
```

Note that it didn't go away. The reason is that there is still a hard reference to it, **/can**. NeWS has a side effect of garbage collection of canvases: it removes them from the display. If we destroy our reference, the canvas disappears, but with no message:

```
    /can null def
```

There is no message because there are no soft references outstanding, thus the object goes through standard NeWS garbage collection. Now make another canvas, and then simply remove our reference.

```
    MakeCan
    /can null def
I'm going away!
```

This results in the **/Obsolete** message. This occurs because the canvas, being still active, has an event manager which has a soft reference to the canvas. By our setting our hard reference to null, the only remaining references are soft event manager references, and the **/Obsolete** event is generated, causing **/obsolete** to be sent to the canvas.

3.7 Class Interfaces to Client Side Services

One of the more creative uses of classes in TNT has been using classes to provide an object oriented interface to services far too heavy-weight to be implemented in PostScript. This is done by reversing the typical window system client-server model: the NeWS display server invokes a client-side "daemon" process that listens for requests from NeWS using the NeWS Wire Service. The Wire Service package is a TNT library using the CPS client-server communication package for NeWS[10]

The client process is not a typical window system application; rather it is designed to listen for client requests, responding by managing client canvases. A simple PostScript class is built with methods which communicate with this service. This provides a light weight PostScript class-based interface to the service. The first instance created for the class causes the client side daemon to initialize and start up. The implementation of the client service may be any language; typically C and C++, but any language with a NeWS Wire Service interface may be used.

The first application of this idea is Jot[8], a C based text package. Jot evolved from the Ched editor described in the NeWS Book[6]. It originally was packaged as a C callable library. This proved to be clumsy for rapid prototyping within the TNT group, however; there were three machine architectures to compile for and it was awkward to re-compile programs for simple applications. The solution was to provide a second interface to the Jot library: a textserver client using the Jot library, but as a text daemon listening for requests from **ClassText**, a subclass of **ClassCanvas** which implements its methods as Wire Service calls to the textserver textserver.

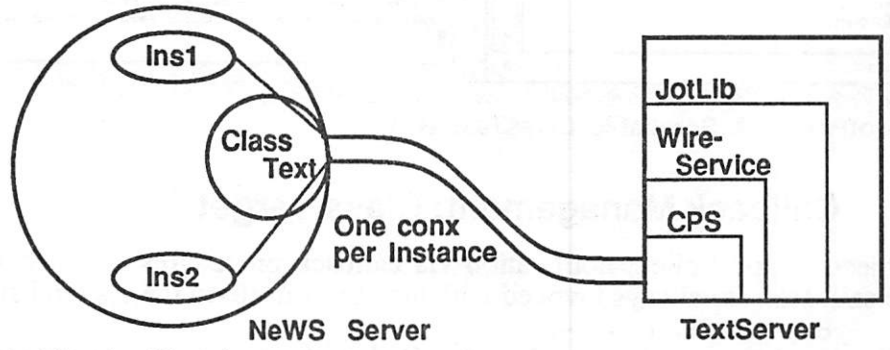
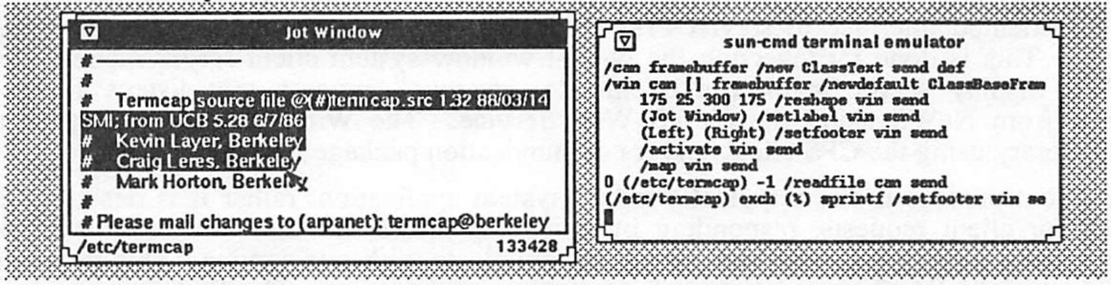


Figure 16: The Jot Model

The use of Jot from PostScript is similar to other **ClassCanvas** instances:

```
/can framebuffer /new ClassText send def
/win can [] framebuffer /newdefault ClassBaseFrame send def
    100 100 400 200 /reshape win send
    (Jot Window) /setlabel win send
    (Left) (Right) /setfooter win send
    /activate win send
    /map win send
0 (/etc/termcap) -1 /readfile can send
(/etc/termcap) exch (%) sprintf /setfooter win send
```

The first statement creates a new instance of a Jot canvas, **ClassText**. The next few lines create and initialize a **ClassBaseFrame** instance with the **ClassText** instance as its client. The last two lines read in /etc/termcap and set the frame's footer to be the file name and byte size.



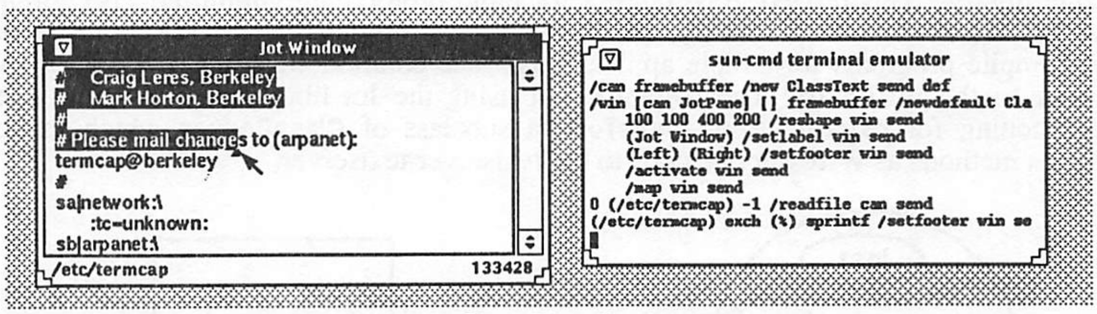
```
#
#
#    Termcap source file @(#)tenncap.src 1.32 88/03/14
SMI; from UCB 5.28 6/7/86
#    Kevin Layer, Berkeley
#    Craig Leres, Berkeley
#    Mark Horton, Berkeley
#
# Please mail changes to (arpanet): termcap@berkeley
/etc/termcap                                   133428
```

```
/can framebuffer /new ClassText send def
/win can [] framebuffer /newdefault ClassBaseFram
    175 25 300 175 /reshape win send
    (Jot Window) /setlabel win send
    (Left) (Right) /setfooter win send
    /activate win send
    /map win send
0 (/etc/termcap) -1 /readfile can send
(/etc/termcap) exch (%) sprintf /setfooter win se
```

Screen 8: ClassText, a Class based Interface to a Client Service

A scrollable pane is also provided which nests the **ClassText** instance within a canvas with scrollbars. This is achieved by simply replacing the first two lines above with

```
/can framebuffer /new ClassText send def
/win [can JotPane] [] framebuffer /newdefault ClassBaseFrame send def
```



```
#    Craig Leres, Berkeley
#    Mark Horton, Berkeley
#
# Please mail changes to (arpanet):
termcap@berkeley
#
sa|network:\
    :tc=unknown:
sb|arpanet:\
/etc/termcap                                   133428
```

```
/can framebuffer /new ClassText send def
/win [can JotPane] [] framebuffer /newdefault Cla
    100 100 400 200 /reshape win send
    (Jot Window) /setlabel win send
    (Left) (Right) /setfooter win send
    /activate win send
    /map win send
0 (/etc/termcap) -1 /readfile can send
(/etc/termcap) exch (%) sprintf /setfooter win se
```

Screen 9: JotPane - A Scrollable ClassText Bag

3.8 Callback Management: Class Target

TNT UI objects support client notification via callback procedures installed in the object. These callbacks are always invoked with the object itself on the operand stack:

```
obj callback => -.
```

Button callbacks, for example, are always called with the button object itself on the stack.

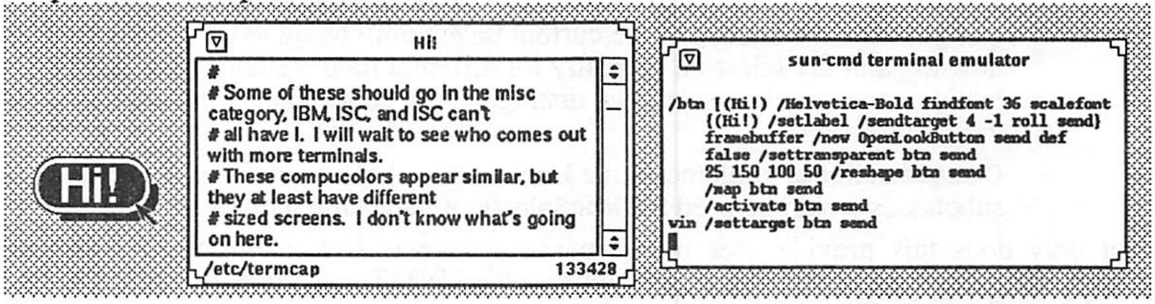
Generally the callback does not want to change the UI object, however. Instead it wants to manage another "target" object. The obvious way to allow this is to install a soft reference to that target in the UI object. Unfortunately softening the target does not completely solve the problem: the **/obsolete** message will be sent to the target, not the UI object. We'd like to avoid requiring the target knowing it is bound to a UI object. In other words, we want obsolescence of the target to send a message to the associated UI object(s).

The solution is **ClassTarget**. **ClassTarget** implements the simplest version of external reference obsolescence management. It does so by creating a separate set of interests in the obsolescence of its target clients. When a target is installed, it is softened and an **/Obsolete** interest is added to a global target event manager. The interest triggers a procedure which tells the UI object to reset its target to **null**.

As an example of target usage, we'll place a button on the framebuffer that will cause the Jot window created above to repaint itself.

```
/btn [(Hi!) /Helvetica-Bold findfont 36 scalefont]
    {(Hi!) /setlabel /sendtarget 4 -1 roll send}
    framebuffer /new OpenLookButton send def

    false /settransparent btn send
    25 150 100 50 /reshape btn send
    /map btn send
    /activate btn send
win /settarget btn send
```

The first three lines create the button with a callback that sends **/damage** to its target. The rest installs the button on the framebuffer, and sets the button's target to be the previous example's window.



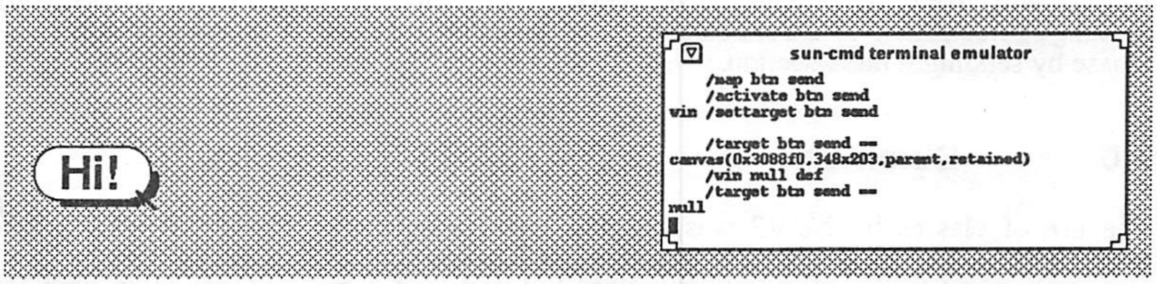
```
#
# Some of these should go in the misc
category, IBM, ISC, and ISC can't
# all have I.  I will wait to see who comes out
with more terminals.
# These compucolors appear similar, but
they at least have different
# sized screens.  I don't know what's going
on here.
```

```
/btn [(Hi!) /Helvetica-Bold findfont 36 scalefont
    {(Hi!) /setlabel /sendtarget 4 -1 roll send}
    framebuffer /new OpenLookButton send def
    false /settransparent btn send
    25 150 100 50 /reshape btn send
    /map btn send
    /activate btn send
win /settarget btn send
```

Screen 10: ClassTarget Usage

To see the obsolescence management, we'll set our **win** handle to **null**. The window disappears because it receives an **/Obsolete** event. The button's target is also reset to **null**:

```
    /target btn send ==
canvas(0x308820,305x199,parent,retained)
    /win null def
    /target btn send ==
null
```



```
    /map btn send
    /activate btn send
win /settarget btn send

    /target btn send ==
canvas(0x3088f0,348x203,parent,retained)
    /win null def
    /target btn send ==
null
```

Screen 11: ClassTarget Garbage Collection

3.9 Look & Feel Independent Selection Model

A prime goal in modern window toolkits is providing a client interface independent of the particular Look and Feel in use. Thus changing from click-to-type to follow-cursor should entail no change in the client application. Converting between left and right handed use of the mouse, and between one, two, and three buttons should also be transparent to the client.

Although complete UI independence is impossible because of potentially new semantic content of user interface, reasonable abstraction is possible in practice. An area in which TNT has been particularly successful is selections[4]. It has successfully built clients that are insensitive to changes between Mac style wipe-through selections and OpenLook style point-and-adjust selections, for example. TNT implements selections as three classes: **ClassSelection, ClassSelectable,** and **ClassSelectionUI.**

- **ClassSelection** describes the selected *data*. It manages the selected data content and transfer among clients. It negotiates how clients choose between data types such as ASCII text or PostScript graphics, and whether or not the transfer is by files or strings.
- **ClassSelectable** is concerned with the *rendering* of the selection data. It abstracts user interface actions into seven standard requests. It negotiates between the client and the current **SelectionUI** being used. It knows how to paint the selection correctly for different modes such as primary selection and secondary selection, distinguishing between pending delete versus insertion.
- **ClassSelectionUI** implements the Look and Feel *state machine*. Separate subclasses would be used for **MacSelectionUI** and **OpenLookSelectionUI.**

Not only does this provide user interface independence, it also promotes sharing of code among similar clients. Thus both Jot and the TNT TextControl selections are derived from the same **ClassSelectable.**

The user's preferred **SelectionUI** is activated during NeWS initialization. Clients supporting selections instantiate their subclass of **ClassSelectable,** which in turn registers them with the **SelectionUI.** Subsequent UI-client interaction in making a selection occurs via sends from the **SelectionUI** to the client's **Selectable.** A selection is initialized by the **SelectionUI** sending **/newselection** to the client's **Selectable,** which in turn sends a **/new** to its associated **Selection.** This **Selection** instance lives for the duration of the selection, and is then discarded. The **SelectionUI** will send the commands: **/selectat /adjustto /dragat /dragto /attachinsertionpoint** to the Selectable. It also will make the **/inselection?** query. The selection database is accessed by sending **/getselection** to **ClassSelection.** A **Selection** instance is inserted into the database by sending it **/setselection.**

4.0 Summary

The use of classes by NeWS was, by and large, a defensive posture; it was simply the only way we could see to successfully add packaging to PostScript! The dictionary representation for classes in instances has provided a graceful migration path for adding classes to PostScript. All of the classing system can easily be implemented in PostScript itself; it needs no interpreter enhancement.

We have been quite pleased by the results of increasing the use of classes in TNT. Use of classes in window programming is clearly a good fit. Integrating the container hierarchy into the class hierarchy provides alternative inheritance styles. Merging the NeWS objects into instances via **/newmagic** vastly simplifies the interaction between the window platform, NeWS, and its toolkit, TNT. We even found it possible and reasonable to implement an X window manager in TNT.

The new directions we are pursuing are printing (both driving printers and application document printing), internationalization, and Jot-like client daemons for other editors. One interesting direction being investigated is an environment not unlike an object oriented HyperCard using PostScript as an interpretive shell to glue the individual components. We'd like to use the P-Shell to do for window components what the Bourne shell does for Unix commands.

The primary measure of success for NeWS classes is that we now spend far more time designing than we do implementing! The final measure of success is simply this: those using the system smile more often than not!

5.0 References

[1] Adobe Systems Incorporated, *PostScript Language Reference Manual*, Addison Wesley, (1985)

[2] Owen Densmore, *Networking NeWS* chapter 10 of *Unix Networking*; Kochan and Wood, ed., Haden Books, (1989)

[3] Owen Densmore, *Object Oriented Programming in NeWS*, Monterey USENIX Graphics Workshop (1986)

[4] Jerry Farrell, *Client-UI Separation for Selections in the NeWS Toolkit*, submitted to CHI-90, (1989)

[5] Adele Goldberg, David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley (1983)

[6] James Gosling, David Rosenthal, Michelle Arden, David Lavallee, *The NeWS Book*, Springer-Verlag (1989)

[7] Stuart Marks, *The pswm X Window Manager*, Sun Microsystems internal presentation (1989)

[8] Jonathan Payne, *Jot: Jonathan's Own Text*, private communication, (1989)

[9] Dave Singer, Rafael Bracho, *Schlumberger Rewrite of Class.ps*, private communication (1988)

[10] *NDE Technical Reference Manual*, Sun Microsystems (1989)

[11] *The X11/NeWS Reference Manual Set*, Sun Microsystems (1989)

Visual Programming with Arachne

John Danskin and Sally Nathan Rosenthal

Digital Equipment Corp. 100 Hamilton Ave. Palo Alto CA. 94301

{jmd,sallyr}@decwrl.dec.com

ABSTRACT

Arachne is a visual, connection based dataflow language, designed for generation and maintainence of test cases and demos for a 3D graphics library on a high performance graphics workstation. In order to carry out these tasks, Arachne is optimized to support 3D modeling, animation, and user interface construction. Models are conveniently built as dataflow expressions. Abstraction conserves screen space and promotes hierarchical design. Animation is made possible by the exposure of time as an explicit factor in program execution.

1. Introduction

We are developing Arachne as a way to generate and maintain tests and demos for a 3D graphics library running on a high performance workstation. In this role Arachne provides a consistent interface between disparate components, allows users to connect these elements interactively, provides ways to interactively change options in hierarchically arranged tests, and provides a way to save these configurations. Arachne is not intended to be a general purpose programming language.

An Arachne program is a collection of built-in boxes with input and output ports which can be connected interactively. Data flows from box to box through the ports. Data flow is clocked; one cycle elapses between the time that a box receives a new input, and the time the filtered data is presented to any downstream boxes. Boxes transform data and may display data-dependent views (such as pictures of 3D data). Boxes can be grouped into *abstractions* in order to save screen space and to provide for hierarchical design.

The dataflow model embraced by Arachne is fundamentally different from constraint-based graphical programming exemplified by ThingLab [Bornin81a] [Bornin86a], where data can flow bidirectionally, and process (time) is not exposed. In constraint based systems, a problem is fed to a program which thinks about the problem for a while and then returns an answer (solves the constraints). This process can be interactive in that the initial state of the system may be varied by the user at run time. In Arachne, program execution is visible. Exposing time permits the use of cyclic dependencies to implement animations.

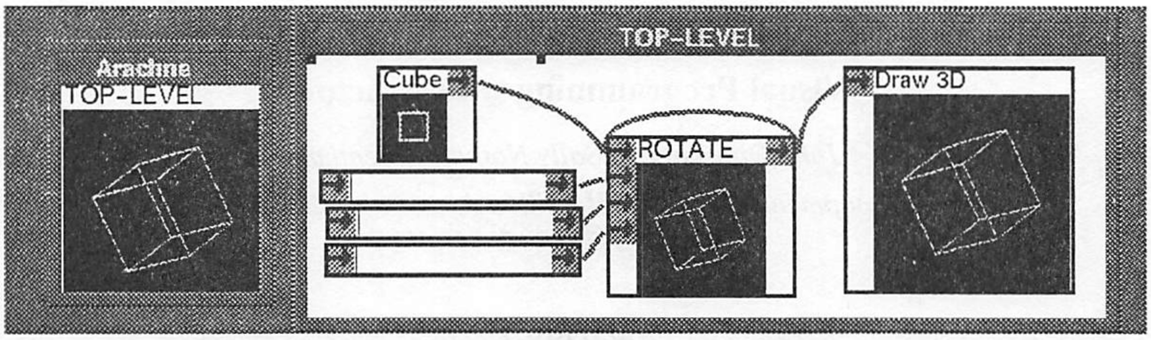


Figure 1. A Rotating Cube

Figure 1 shows an Arachne program implementing a rotating cube. Wiring the output of **ROTATE** into its input causes the original cube to be successively rotated. **CUBE** will only send its output when the connection is made.

Arachne is similar to, and inspired by, Conman [Haeber88a], but has several new features: the ability to abstract, a well-defined execution model, automatic connection routing, program saving, and low overhead message passing. Arachne is even closer to Fabrik [Ludolf88a], but differs in its emphasis on 3D graphics, the execution model, and recursion. Published work on Fabrik does not discuss the execution model, nor does it discuss looping or conditional constructs.

Sections 2 and 3 discuss the problems that Arachne was built to address: testing 3D graphics libraries, and generating Demos for graphics workstations. Section 4 outlines Arachne functionality and structure. In sections 5 and 6 we show how Arachne can be applied to testing and demo generation. Section 7 explains the examples in Figures 2 and 3. Look and feel issues are discussed in section 8. Section 9 gives details of our current implementation. Section 10 talks about problems with Arachne. Section 11 is about future work, and Section 12 concludes.

2. Testing a 3D Graphics Library

A 3D graphics library is a large collection of primitive calls, each with a large collection of option variables, each with several possible values. Generation and organization of testing scripts is a major problem. Watching pictures go by, matching them to expected results and rendering parameters is at best a tedious task. Maintaining old testing scripts for regression testing is nearly impossible as system characteristics change, and old scripts are cannibalized to make new ones. Interactive testing with dynamics reveals many bugs that static testing misses. Unfortunately, these bugs may be difficult to reproduce, especially when system response is slowed by debugging options.

Arachne provides a framework for grouping related tests, and viewing them simulataneously. Interactive testing in Arachne is repeatable, even under extreme debugging situations.

3. Generating Demos

Demos should be interactive when possible. They should look like real applications, but must be created with far fewer resources. To fully demonstrate a graphics workstation's potential, demos should at least span the application space from mechanical CAD to volume visualization to compute-intensive applications. Additionally, demos should be clear, easy to use, and should be able to run without user interaction.

Arachne's user interface construction capabilities simplify turning algorithms into demos. The connection motif allows users to connect algorithms together to create new demos interactively. Recording facilities and the exposure of time make it easy to build animated demos and short movies.

4. Language Components

4.1. Programs

An Arachne program is a single top level *abstraction.* If this *abstraction* is *closed,* then an Arachne program presents a single window with a user defined interface. If the *abstraction* is *open*, then another window presents an editable view of the Arachne program. *Abstractions* can contain boxes and other *abstractions*. Boxes and abstractions have input and output ports. Input ports are on the left; output ports are on the right. Boxes and *abstractions* may have any number of either type of port. An input port may be connected to any number of output ports and vice versa (fan in and fan out). Connections are displayed as 'wires' between ports.

4.1.1. Ports

All data is transferred to and from ports. Input ports latch incoming data so that boxes can fire when a single input (one of possibly many) is changed. The effect of multiple pieces of data arriving at a port during a single cycle is undefined.

4.1.2. Boxes

Boxes are Arachne units of functionality. They correspond to built-in operators in textual languages. Boxes may have static displays, or data-dependent displays (for instance, they may display 3D data). Some boxes are interactive. For example, a slider outputs numbers in the range zero to one as the user moves the cursor along the box. Boxes receiving incorrectly typed data flag it as invalid and are quiescent until the port is disconnected, or new valid data arrive.

4.1.3. Abstractions

Abstractions are the facility in Arachne that allow for the construction of interesting programs. Without *abstractions,* complexity becomes overwhelming with only a few primitives, and programming halts. The editable view of an *abstraction* is a window. This is the only window in which program editing can occur. While the *abstraction* is *open* (there is an editable view), the contents of the *display* are undefined. When the *abstraction* is *closed* the editable view disappears and the *display* is updated. We discuss displays in more detail below. Connections are not allowed to span *abstractions.* To send data into *abstractions,* input ports or radio transmitters are used. To receive data from an *abstraction*, output ports are used.

The closest analogy to the *abstraction* in textual programming is the first class procedure with closure [Rees86a]. A better analogy for *abstractions* is hardware, since they may have persistent state and may be active concurrently with other parts of the program. Think of the *abstraction* as an autonomous functional unit. A recursive *abstraction* implements a potentially infinite amount of hardware.

An *abstraction* used in a program acts exactly as though the boxes in the *abstraction* had been used. There is no delay across *abstraction* boundaries. Since *abstractions* may have their own state, they may implement functions requiring persistent state.

4.1.3.1. The Display

An *abstraction* has two visible representations: the editable view and the display. The editable view is a resizable window into which boxes can be copied, and in which arbitrary editing operations take place. One of the editing operations is to specify the display. The display is the rectangular area of the editable view which is visible in the parent *abstraction* where the child *abstraction* is instanced. If this rectangle includes interactive components, they are available for interaction in the display. Since the interactive components can be wired arbitrarily, this allows a primitive interface construction facility. This mechanism has been previously described in [Ludolf88a]. A better, but somewhat more complex mechanism was developed in [Smith88a].

4.1.4. Execution Model

A predictable execution model allows construction of loops and conditionals. There are visual languages without these features [Upson89a, Haeber88a], limiting predictable programs to acyclic graphs. AVS [Upson89a] takes advantage of this feature, providing more efficient evaluation of acyclic graphs than either Conman or Arachne. Other languages have implemented various other models. The best execution model for general dataflow programming may be that used in Show and Tell [Kimura86a, Kimura86b]. There, puzzles are completed in a predictable way (the only way), and consistent/inconsistent puzzles enable further computation. Arachne implements a much lower-level mechanism that exposes time in a useful way. In Arachne, each box executes in a consistent amount of time. All implemented boxes execute in one cycle.

Arachne's exposed program execution makes intermediate program steps visible. This makes animation easier to create. More sophisticated execution models usually limit a given box or wire to a single value. They concentrate on making that value predictable. Arachne allows wires to assume multiple values through time, but allows the user to predict the value at a particular time. Figure 1 (above) is an example of this ability.

4.1.5. Radio Boxes

Radio boxes implement 'wireless' connections. They are a mechanism for reducing the amount of wiring and the number of ports on *abstractions*. There are two kinds of radio box: transmitter and receiver. Both are 'tuned' by a text string (the call letters). The transmitter broadcasts whatever messages it receives on its input port to the channel selected by its call letters. Receivers receive messages broadcast to their channel and send them out their output port. The scope of a transmitter is its *abstraction* and all *abstractions* included by that *abstraction*. Local transmitters have precedence over enclosing transmitters. Only one transmitter can be tuned to a given frequency in a single *abstraction*. In a hierarchy of *abstractions*, one should use Radio boxes to send messages which are common among child *abstractions*. Connections must be used for other messages.

4.1.6. Saving and Restoring

Arachne saves the entire running state of the program (including messages sent but not received). Thus, programs can be saved and restored with no loss of information. Although Arachne's clock can be stopped to halt execution, Arachne makes no distinction between running and non-running programs.

4.1.6.1. Copying Abstractions

The ability to create *abstractions* is an efficient way to save screen space. However, *abstractions* are much more useful, especially for user interface construction, if they can be duplicated. There are two ways to duplicate an *abstraction*: deep and shallow. A deep copy makes a total copy of the (running) *abstraction*, including loop state. A shallow copy is just like a deep copy, except that the shallow copy remembers where it came from. Making shallow copies creates new members of an equivalence class. Whenever one of the members of a shallow copy equivalence class is edited, all other members are updated. This has the side effect of resetting the state of all other members to the state of the edited member.

4.1.6.2. Recursion

Since we can make a shallow copy of an *abstraction*, the question arises, what if a shallow copy is moved into the editable view of a member of the copy's equivalence class? Arachne generates a recursive *abstraction*. In implementation, we are careful not to try to instance all levels of recursion: just those which might be visible, and (lazily) those which receive messages. Additionally, the system keeps track of recursion levels and enforces a maximum level of recursion to prevent the user from inadvertently crashing the system.

4.2. Built-in Abstractions

The flavor of a language is largely determined by what is built in and what is left out. Mostly we left out things the user can build, like user interfaces for transformations, and color generation. While any function we could implement in Arachne could be more efficiently implemented in Arachne's implementation language Scheme, we prefer to generate a library of Arachne *abstractions,* since the number of such interfaces could be large. We provide enough functions so that users can build their own interfaces.

The **event recorder** captures events as they arrive. Events can then be played back at the user's convenience. To facilitate graphics library debugging, the recorder supports random access, and can play in either direction. To support animation, the recorder supports dubbing, insertion, and has an endless loop facility with either wrap-around or reverse at boundaries. Playback can be initiated by a message on a connection, which allows synchronization of multiple recorders. The event recorder was previously implemented in [Haeber88a]. The **animation recorder** is similar to the generic event recorder, but has some optimizations, such as frame rate control, for generating and viewing animated sequences. **Sliders** can be grouped in abstractions to produce useful tuple sliders, such as XYZ positions or RGB color values. **Number** and **String** boxes allow keyboard input of these values. Among our **3D primitives** are: cubes, cylinders, cones, spheres, tori, polygon files, generalized cylinders, free form deformations, and NURB patches. Along with some free form deformations, we explicitly support the following **transformations:** rotation, translation, scaling. A special **3D rendering** box displays graphics and exposes many options for graphics library testing.

5. Using Arachne for Testing

We created a 3D drawing box that allows the user to set any implemented PEX [Rost89a] options. Options are settable through ports to allow program control. Multiple tests can be quickly built and saved for later use.

One way to save time and capitalize on human visual capabilities is to display several similar tests simultaneously. If the data is the same, but rendering options are different, then the task of finding an option which fails completely is considerably simplified. In Arachne, one may create an array of related tests and arrange them in a grid inside an *abstraction.* Each test is an *abstraction* itself which presents the test picture and a short comment explaining the picture as its external view. Variables with lexical scoping allow many different tests to be run without changing other constants of the test. For instance, we might display a cross-section of the different types of shading allowed by PHIGS+ [AdHoc88a, ISOIEC87a] (there are three orthogonal axes: shading model: none, ambient, diffuse, specular; interpolation mode: flat, linear, dot-product, normal interpolation; lights types: ambient, vector, point, spot). Sixteen *abstractions* display primitives with the sixteen combinations of shading models and interpolation modes, while the tester or an Arache program set up by the tester cycles through the different types of lights.

To reduce the number of wires, the lights, the object to be rendered, and the transformation matrices may be implemented by the tester using radio boxes. Objects and lights can then be changed interactively in all of the windows simultaneously. Event recorders provide the repeatability characteristics so important in interactive testing.

6. Generating Demos in Arachne

Arachne provides a way for different program segments to share a user interface and a database. In order to demonstrate new functionality, a programmer needs only to implement the algorithms involved. Implementing a box with trivial functionality only takes about 30 lines of code, most of which is the same as for other boxes. For instance, one could implement an image processing library by defining a box which reads image files, some boxes implementing imaging operations such as sharpening, edge detection, sum, difference, matting et cetera, and a box to save image files. Then users are free to combine these boxes as they wish.

Arachne's recording and animation facilities make it easy to 'can' complex demos. Save/restore facilities make it possible for the adventurous sales or marketing representative to make his or her own modifications to existing demos. Arachne's pervasive interactive motif makes it almost impossible to create a demo that cannot be interacted with.

7. Examples

7.1. Some Abstractions

The window labeled "TOP-LEVEL" in Figure 2 is the editable view of the top level *abstraction.* Its *display* is not shown. **TOP-LEVEL** contains four boxes.

1). **CUBE** sends a cube out its output port.

2). **Rotate** is an *abstraction* which provides a user interface for specifying rotations.

3). **Draw3D** draws 3D objects arriving on its input port.

4). **Library** is an *abstraction* used to keep other useful *abstractions.*

The output of **CUBE** is connected to the object input of **Rotate.** The output of **Rotate** is connected to the input of **Draw3D.**

Although I/O port functionality is opaque in this figure, the user can display a short note describing port functionality by pressing a mouse button on a port.

The editable view of **Rotate** is shown in the middle of the figure. It has four inputs. The top one accepts objects to be rotated. The others are for programatically setting the slider inputs. Three number boxes initialize the sliders to their middle position. These sliders have been moved since being connected, and the number boxes won't send their data again until they either receive input, or the user causes them to fire by pressing <return> in their active area. **Rot Adjust** is an *abstraction* built to remap slider outputs from the range (0 to 1) to the range (-.5 to .5). We built one instance of **Rot Adjust**, and then duplicated it twice. **ROTATE** (note caps) is a built-in primitive, which accepts a 3d object and rotations about the x, y and z axes. It displays the resulting rotated object and sends it out its port. **ROTATE** is connected to the output port of **Rotate. Rotate**'s *display* is the rectangle covering the active portions of the three sliders.

Library contains another instance of **Rotate,** and a string box which is used here as a comment. As more *abstractions* are added to the library, the user may add more

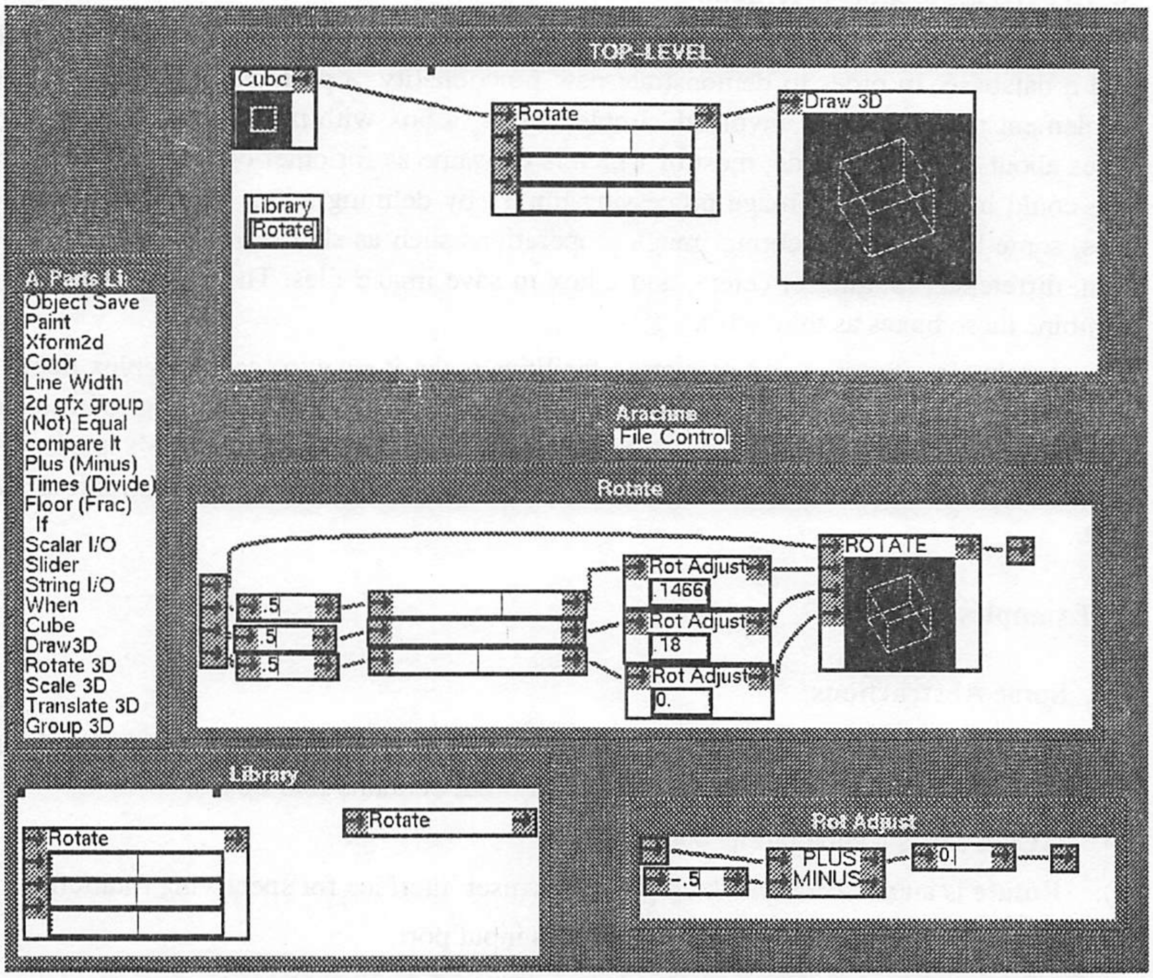


Figure 2. Some *Abstractions*

comments and change the *display* of **Library** so that they are all visible. The instance of **Rotate** visible in the top level *abstraction* was created by duplicating the instance in **Library.**

A.Parts List is a menu of built-in boxes. Clicking on a menu item causes the corresponding box to be created in the top level *abstraction*. The window labeled **Arachne** provides popup menus for functionality such as Save, Quit, Stop, Go, and Single Step.

7.2. A Pyramid

The Arachne program in Figure 3 builds a pyramid with a base of four cubes and another cube on top. Starting at the upper left of the illustration, **Cube** sends a cube to **ROTATE.** Three sliders control rotation about x, y, and z axes. **ROTATE** sends the rotated cube to three **TRANSLATE** boxes. The two boxes at the upper right translate the cube plus and minus one unit in x respectively. These translated cubes are send to **Group3D** at the upper right. This group is sent to the two **TRANSLATE** boxes at the lower left. They translate the cubes plus and minus one unit in y. The two translated pairs

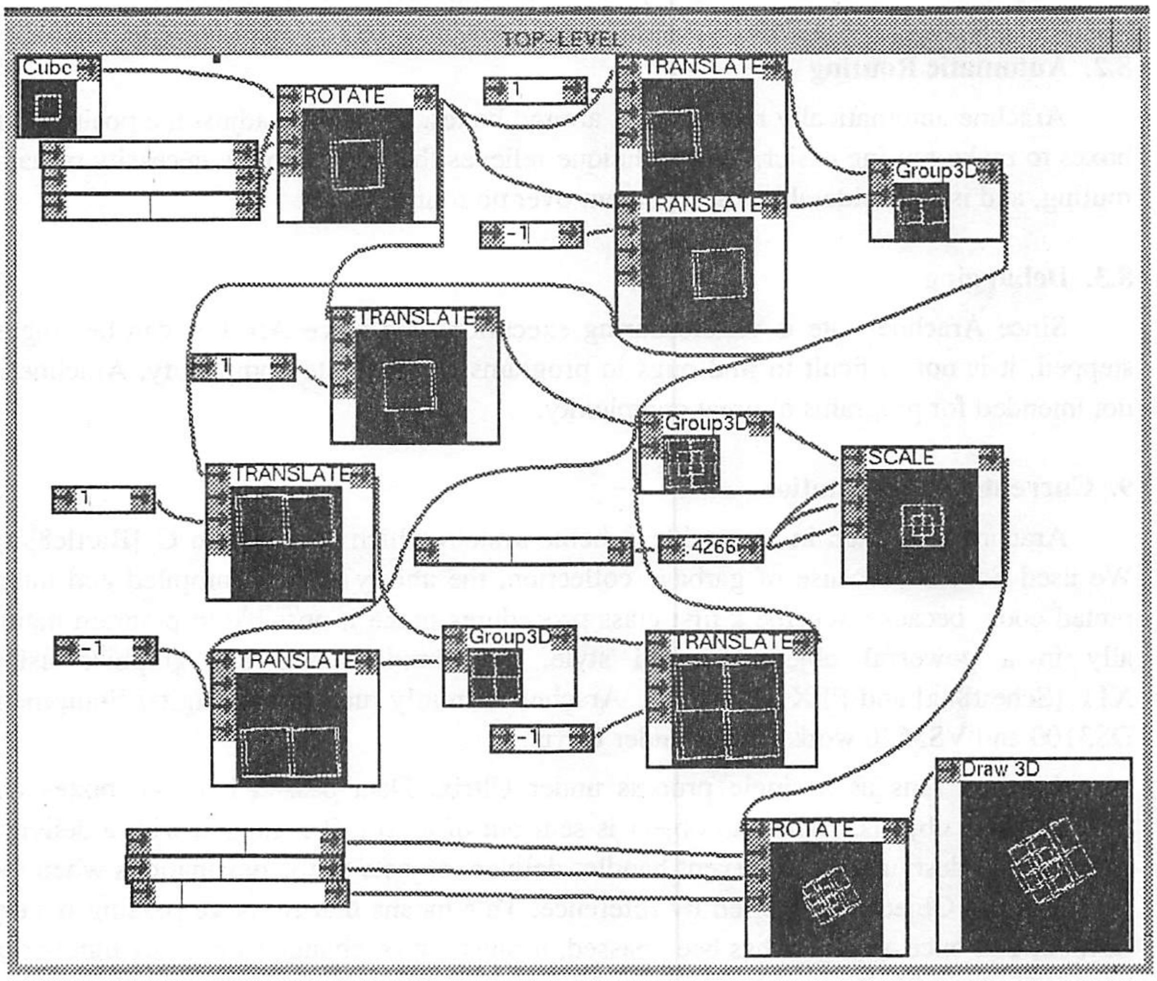


Figure 3. A Pyramid

are grouped to form the pyramid base of four cubes using another **Group3D** box. This base is sent to a **TRANSLATE** box which moves it back one unit. The four cubes are then sent to another **Group3D** box (center right) where they are grouped with a fifth cube which has been translated forward one unit. This conglomeration is getting too big, so we scale it uniformly by about .4 using **SCALE**, controlled by a slider. **SCALE** sends the pyramid to **ROTATE** at the lower right. Rotation of the pyramid is controlled by three sliders. **Draw3D** at the end (lower right) provides a larger view of the pyramid.

8. Look and Feel

8.1. Incremental Execution

A common mode of interaction with some programs is to run the program, discover that some setting or other is not quite right, and run the program again with slightly different settings. This is inefficient in that the variable changed may not have affected the entire calculation, but only some subpart. Nevertheless, the entire calculation is done over again. In Arachne, the user interacts with the running program. Interactive boxes are always active. Only boxes that depend on the interactive box re-execute.

8.2. Automatic Routing

Arachne automatically routes wires around boxes. We do not adjust the positions of boxes to make routing easier. This technique relieves the user from the necessity of hand routing, and is a considerable improvement over no routing at all.

8.3. Debugging

Since Arachne state is visible during execution, and since Arachne can be single-stepped, it is not difficult to find bugs in programs of moderate complexity. Arachne is not intended for programs of great complexity.

9. Current Implementation

Arachne is written in a portable Scheme system which compiles to C [Bartle89a]. We used Scheme because of garbage collection, the ability to mix compiled and interpreted code, because Scheme's first class procedures make it possible to pragram naturally in a powerful object oriented style. We implemented our graphics using X11 [Scheif88a] and PEX [Rost89a]. Arachne currently runs on the Digital Equipment DS3100 and VS3520 workstations under Ultrix.

Arachne runs as a single process under Ultrix. Data passed between boxes are referred to as objects. When an object is sent out of a port, it is tagged with a delivery date and the destinations. An event handler delivers objects to the destinations when the time is right. Objects are passed by reference. This means that message passing is very fast, but that once an object has been passed, it must not be changed since any number of boxes may be using it.

Saving complicates matters considerably. Every object has a *save* method which returns a list which, if interpreted, would create a new object identical to the old one. Every box has a save method which has the same effect. To save itself, a box has to save all of the objects on its input ports. After all of the boxes are saved, the connections are saved. To make it possible to save connections across Arachne invocations, boxes have unique *names*. Each end of a connection is saved as a name, port-number pair. Connections are saved as pairs of ends. Finally, the state of the running program has to be saved. The event handler saves all of the pending messages so that when the program is resumed, an equivalent calendar is rebuilt. A saved Arachne program can be loaded into another running program.

Duplication of boxes is accomplished by interpreting the result of the *save* method and changing the name of the result. Duplication of *abstractions* is harder. First, all of the children of the abstraction are duplicated and given new names. Then they are connected using a mapping from old boxes to new ones. Then they are all moved into the new *abstraction*. Finally, the old event list is duplicated with the new event destinations.

We implement automatic routing using a 2D Bintree. We pad the outlines of the boxes, and add them to the bintree. To route a connection, we flood fill empty space in the bintree, using the start-point as a seed, until we find the area containing the end-point. This generates the path crossing the fewest number of bintree rectangles. After selecting a start point, we route locally through the rectangles by recursively choosing the closest point on the shared boundary of the current rectangle and the next rectangle. This list of points defines a piecewise linear curve which avoids all of the boxes. Unfortunately, while these curves avoid the boxes, they do not avoid each other. We experimented with versions where lines avoided each other, but run-times were not acceptable. Instead of drawing the polyline, we use the points to define a quadratic bspline. Higher order curves have less local control. Therefore, if two bsplines have a few control points in common, they won't necessarily be coincident in that area. But, the higher order the curve, the farther it is from interpolating the control points. We compromised on quadratic bsplines. Using our router written in Scheme we can route 100 connections in about 1/10 seconds on a DecStation 3100.

10. Problems

Connection based visual programming languages give new meaning to the term "spaghetti code". It is easy to build a program with so many wires that it becomes illegible. There are various heuristics for avoiding this. We use automatic routing to assist layout, and radio boxes to cut down on the number of wires. Once a complex piece of code is built, it may be hidden in an *abstraction.*

We are not satisfied with our current routing heuristics. Although automatic routing is a well known hard problem [Dion88a], we are confident that we can achieve higher quality layouts automatically.

It is hard to write programs with non-trivial loops. Synchronizing loop counters with looping data is an absorbing task. We have considered implementing synchronizing primitives described in [Suther89a] but have not felt the need given Arachne's objectives.

Instead of having undefined fan-in semantics, we should disallow fan in. To replace this functionality we would provide a two input one output **MERGE** box. **MERGE** would consistently choose one input over the other in case of a collision.

11. Future Work

We would like to add type checking to Arachne connections. Currently types are checked when data is received. We would rather that it be impossible to connect ports with incompatible types. In the case of polymorphic boxes, it should not be hard to propagate types through the boxes as long as the type system is kept simple.

12. Conclusions

We have presented a visual language which is an effective tool for the target application. Problems with this tool have to do with its suitability for tasks other than those for which it was designed. We believe that the connection metaphor is a valid interface for a wide variety of applications, and that applications built using this metaphor will be easily extensible.

An interface for external programs would make Arachne more modular, and would make it much easier for third parties to extend built-in functionality.

13. Acknowlegements

We would like to thank Allen Akin and Jeff Lane for allowing us to do something useful in an interesting way. Joel Bartlett has provided amazing support for his Scheme to C compiler, and was extremely helpful with this manuscript.

References

Bornin81a.

A. Borning, ''The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory,'' *ACM Transactions On Programming Languages and Systems*, vol. 3, no. 4, ACM, October 1981.

Bornin86a.

A. Borning, ''Defining Constraints Graphically,'' in *Human Factors in Computing Systems: Proceedings SIGCHI'86.*, IEEE Computer Society, Boston, MA., April 1986.

Haeber88a.

P. Haeberli, ''Conman: A Visual Programming Language for Interactive Graphics,'' *Computer Graphics*, vol. 22, pp. 103-111, ACM SIGGRAPH, August, 1988.

Ludolf88a.

F. Ludolf, Y. Chow, D. Ingalls, S. Wallace, and K. Doyle, ''The Fabrik Programming Environment,'' in *1988 IEEE Workshop On Visual Languages*, pp. 222-230, Computer Society Press, October 1988.

Rees86a.

Jonathan Rees, William Clinger, and Et. Al., ''Revised^3 Report on the Algorithmic Language Scheme,'' *SIGPLAN Notices*, vol. 21, pp. 37-79, December 1986.

Smith88a.

D. Smith, ''Visual Programming in the Interface Construction Set,'' in *1988 IEEE Workshop On Visual Languages*, pp. 109-120, Computer Society Press, October 1988.

Upson89a.

C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. vanDam, ''The Application Visualization System: A Computational Environment for Scientific Visualization,'' *IEEE Computer Graphics and Applications*, pp. 30-41, Computer Society Press, July 1989.

Kimura86a.

T. D. Kimura, ''Determinacy of Hierarchical Dataflow Model,'' Technical Report WUSC-86-5, Department of Computer Science, Washington University, March 1986.

Kimura86b.

T. D. Kimura, J. W. Choi, and Jand M. Mack, "A Visual Programming Language for Keyboardless Programming," Technical Report WUSC-86-6, Department of Computer Science, Washington University, June 1986.

Rost89a.

Randi J. Rost, Jeffrey D. Friedberg, and Peter L. Nishimoto, "PEX: A Network-Transparent 3D Graphics System," *IEEE Computer Graphics and Applications*, vol. 9, no. 4, pp. 14-26, July 1989.

AdHoc88a.

AdHoc, *PHIGS+ Functional Description,* Revision 3.0, report issued by ad hoc committee, January 22, 1988.

ISOIEC87a.

ISOIEC, *Programmer's Hierarchical Interactive Graphics System (PHIGS),* ISO/IEC Standard 9592-1:1988(E), International Standards Organization, October 1987.

Bartle89a.

Joel Bartlett, "SCHEME->C a Portable Scheme-to-C Compiler," Digital Equipment Western Research Laboratory Research Report 89/1, Digital, January 1989.

Scheif88a.

Robert W. Scheifler, James Gettys, and Ron Newman, *X Window System C Library and Protocol Reference,* Digital Press, 1988.

Dion88a.

Jeremy Dion, "Fast Printed Circuit Board Routing," Digital Equipment Western Research Laboratory Research Report 88/1, Digital, March 1988.

Suther89a.

Ivan E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, pp. 720-738, ACM, June 1989.

Controlling Virtual Worlds with the Panel Library

David A. Tristram
National Aeronautics and Space Administration
Ames Research Center
Moffett Field, California 94035
dat@nas.nasa.gov

1 Abstract

This talk introduces the Panel Library, a two-dimensional user-interface toolkit, and explores extensions to it that allow the control of three-dimensional virtual worlds. Because the Library is not pixel-oriented, its world-space user-interface elements may be positioned arbitrarily in three-space. This capability is exploited to build three-dimensional control surfaces which may be placed in the same scene as the rendered version of the user's object of study. Issues regarding the mapping of mouse motion to the transformed control surfaces are discussed, as are surface visibility, rendering problems and performance. Finally, implications for future interfaces to virtual worlds are noted.

2 The Panel Library

The Panel Library is a flexible toolkit for building interactive graphics applications on Silicon Graphics Iris workstations. The users of the Panel Library use a mouse to manipulate graphic user-interface elements drawn on the workstation screen, and receive instant feedback on their input. Panel Library programs use functions to create windows that are used as "control panels." Other functions are used to create user interface elements which are then placed on the panels. After they are created, any aspect of an interface element can be modified by updating the corresponding field in its descriptor. The Library immediately and automatically updates the graphical representation of the element on the screen to match its new position, size, or value.

3 Nomenclature

User interfaces built using the Panel Library consist of mouse-sensitive graphic control elements, called "actuators", that appear in one or more control "panels." Panels are implemented as window

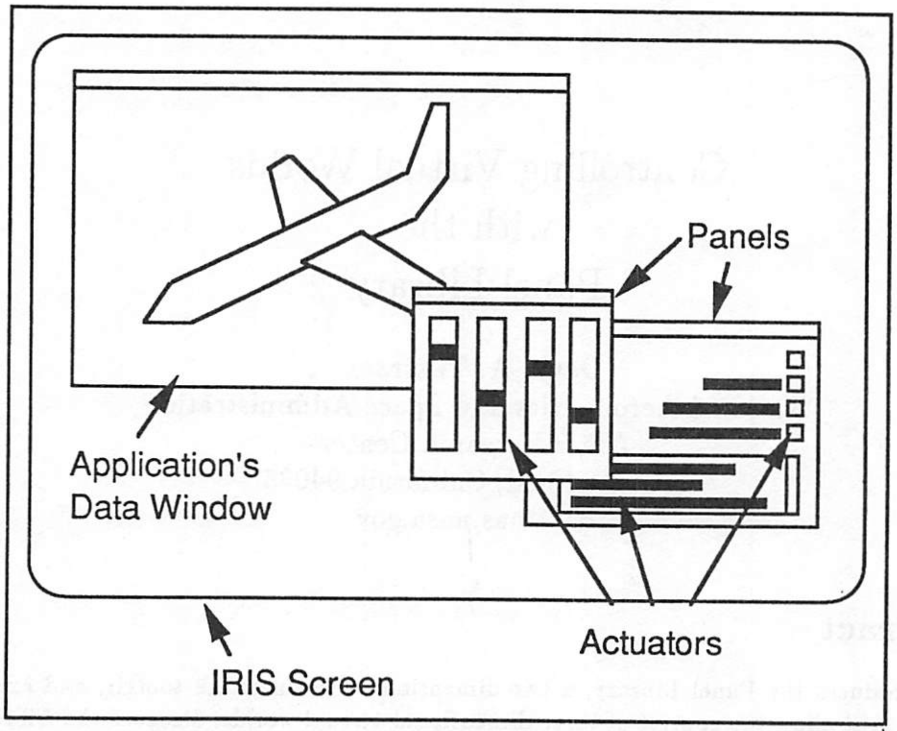


Figure 1: Elements of Panel Library interface.

manager windows, which the user can move and rescale in a consistent, application-independent way using the mouse. See figure 1.

Actuators may be classified as either "simple" or "compound." Simple actuators implement basic control panel functions like Buttons, Sliders, Dials, and Typeins. Compound actuators serve to group actuators by accepting subactuators, thus allowing a hierarchical relationship. Examples of compound actuators are Frames, Menus, Icons, and Cycles.

Each actuator and panel may have associated with it "action functions" which are called when the mouse button changes state, and while the mouse button is held down. Actuators and panels also have a number of instance variables which contain mode and state information.

In addition to the control panels, a Panel Library application may have one or more user Data Windows. These are the graphics ports as supported by the Iris Graphics Library (GL) where the user's model or scientific data set is rendered.

4 Coordinate Systems

Panels are created by the Library with a default orthographic coordinate system which scales panel world space units to screen space pixels. Actuators are dimensioned using floating point world space coordinates. The dimensions are chosen so that actuators laid out on integer or half-integer grids appear aesthetically balanced. That is, Sliders, for example, are a little less than one unit wide, so that a row of them may be placed at x=0,1,2,.. When they are created, panels are sized so that all actuators that have been added to them will fit at the the default scale factor. The automatic sizing of panels assures actuators of the same type appear the same size on different panels.

A compound actuator may contain a number of subactuators. Depending on the semantics imposed by the compound actuator, none, one, or more of its subactuators may be drawn at a given time. Compound actuators provide an independent coordinate system that may be offset or scaled.

Frame, the generic grouping actuator, has modes in which it sizes itself to accommodate its subactuators, or scales its contents to limit its overall size. In addition, Frame provides a second offset that allows repositioning or scrolling of its contents. In the Library, Frames are implemented as independent Iris GL viewports which provide hardware clipping.

5 Three-dimensional Extensions

The interface library as described above implements a complete, traditional two-dimensional graphical user interface. However extensions may be made to provide more direct control over three-dimensional systems. The first of these uses the Viewframe to place the two-dimensional control panels into three-space.

The Viewframe actuator performs the same grouping functions as the Frame. In addition, Viewframe includes a transformation by an arbitrary three-dimensional modeling matrix before drawing its contents. This transformation lets the programmer place the rotated, scaled, and translated contents of the Viewframe in any relationship to the viewer and the rest of the user interface elements.

Mouse position information must be transformed into the coordinate system of the Viewframe to allow user manipulation of its contents. This transformation is accomplished by forming the inverse of the viewing transform. The mouse position becomes a line in the coordinate system of the Viewframe. Since the Viewframe is a two-dimensional rectangle, the line is simply intersected with the plane z=0. The x and y coordinates of this point are the mouse position in the two-dimensional Viewframe coordinate system.

In the two-dimensional Panel Library paradigm, the behavior of overlapping actuators is undefined. With three-dimensional actuators and in the anticipation of changing user viewpoints, overlapping and intersecting actuators must be handled consistently. In the default case, it can be assumed that the actuator nearest the user along the world space projection of the mouse position is the intended selection target.

6 Rendering Problems

On the Iris, depth- or z-buffering is used to provide hidden surface removal. Rendering actuators while z-buffering requires some changes in the rendering scheme. Most actuators possess a moving or changing element over a neutral background. If these elements are all drawn at the same z value, z-buffer priority will be ambiguous. To avoid the visual crosshatching symptom of z-buffer ambi-

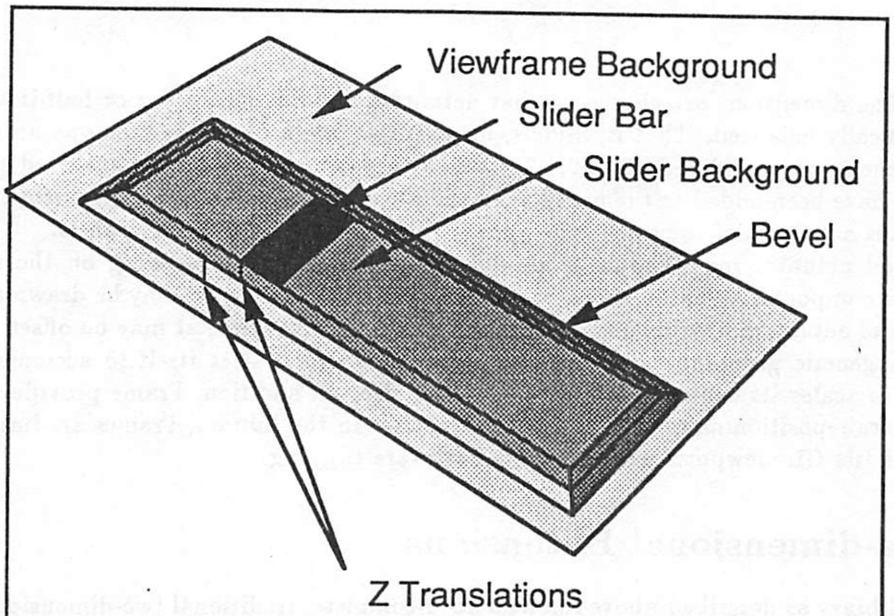


Figure 2: Adjusted three-dimensional slider geometry allows rendering with Z buffer.

guity, active portions of the actuator are translated above the surface of the background. Beveled outline surfaces are used to keep the raised portions of the actuator from drawing onto unobscured background. See figure 2. Although these translations allow actuators to be drawn correctly once, the z-buffer has further impact on animation.

When drawing its two-dimensional actuators, the Library can avoid redrawing unchanging portions of the screen by updating only the (single) currently active actuator. This efficiency measure is important in keeping frame rate high. Because two dimensional actuators do not overlap, and have a well-defined background, they can be rendered correctly using a painter's algorithm. In the three-dimensional case, however, the painter's algorithm fails because the background will not overdraw the moving portions of the actuator that have been translated toward the viewer to avoid z-buffer ambiguities. Clearing the z-buffer and redrawing the actuator allows animation but loses rejection of hidden surfaces possibly obscured by other actuators.

There are at least two solutions to the problem of rendering groups of three-dimensional actuators. One is to simply render all parts of all actuators for each screen update. For simple control interfaces this is possible with current hardware. The second solution renders a subset of the control panel by detecting all actuators that intersect the active actuator as seen from the current viewpoint. Actuators that intersect any member of that set are added to the list until there are no further intersections. Using the Iris zdraw mode, the z-buffer is cleared inside the union of their projections. All actuators in the set are then rendered normally. See figure 3. Selecting an active subset of active actuators may be advantageous when CPU performance is better than graphics or in the case of extremely complicated control surfaces. The set of intersecting actuators needs only be recalculated

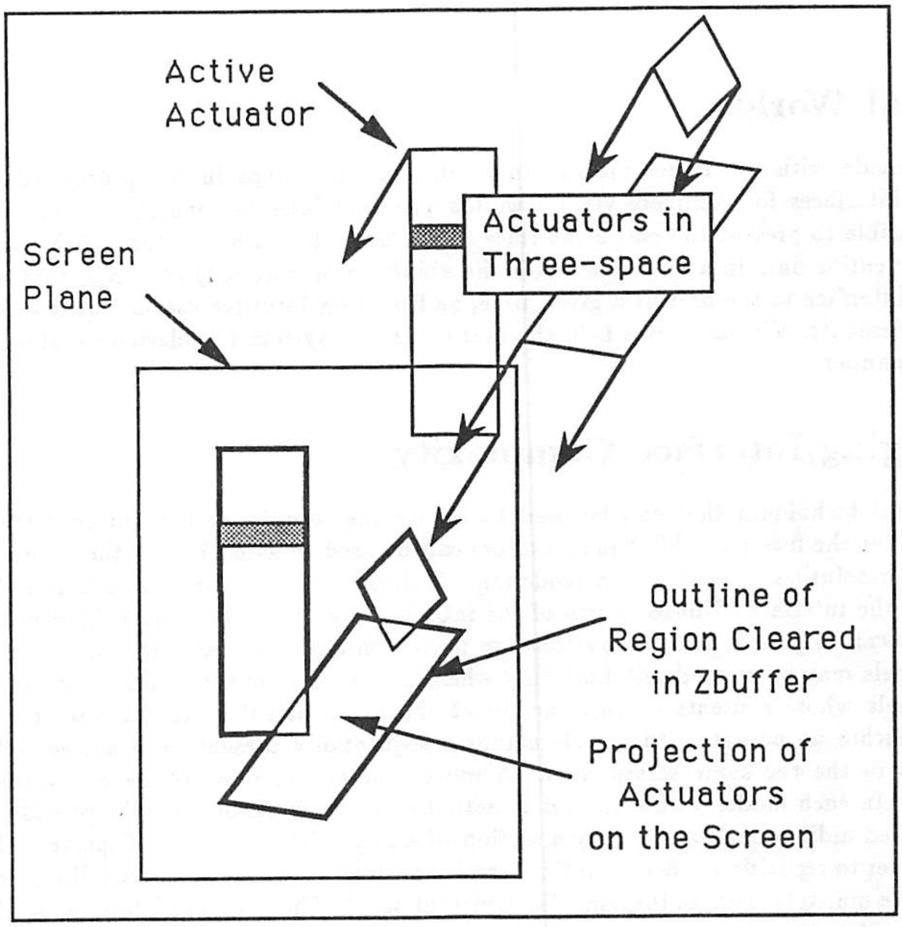


Figure 3: Animation in three dimensions requires redrawing all overlapping actuators.

when the viewpoint is updated or when a new actuator becomes active.

There are also techniques for layering multiple geometries in the same z location that combine drawing into the z-buffer and using the writemasks. Although these techniques can ameliorate the problem of z-buffer ambiguity without distorting the geometry of the actuators, they require a depth-sort of rendered surfaces and do not solve the case of intersecting actuators.

It should be mentioned that in the two-dimensional case, a Frame with its associated viewport provides hardware clipping to its boundary. For Viewframe, no such hardware support exists, so that implementors of actuators must exercise special care to avoid drawing outside their bounding rectangles.

7 Virtual Worlds

The progress made with the Panel Library takes the first few steps in the process of building workable user interfaces for complete virtual worlds. As interfaces become more complicated, it becomes impossible to present the entire interface to the user at all times. Much of the attraction of presenting scientific data in a virtual world is the ability to present only the important or active portion of the interface to the user at a given time, and to allow intuitive user-directed selection of user interface elements. Virtual worlds help the user utilize the system's rendering capability in the most efficient manner.

8 Managing Interface Complexity

There are several techniques that can be used to reduce the complexity and improve the screen resolution used for the interface. Multiple monitors can be used to avoid having the interface steal valuable screen resolution from scientific rendering. Multiple control panel windows may be used to hide part of the interface. Unused parts of the interface are "pushed" away behind others. To allow random recall of panels, selector buttons are placed on a master panel that is never hidden. Momentary panels may be created automatically when certain parts of the interface are activated.

Control panels whose contents change can direct the user's attention to the next part of his or her task. Within an actuator, the Cycle actuator sequentially presents alternative collections of subactuators in the the same screen area. A more general technique is the establishment of panel "modes." In each mode, a different set of actuators is made visible. Scrolls provide a more "physically"-based hiding mechanism; only a section of a large virtual panel is displayed while the user moves a slider to reposition it behind a fixed scrolling window. A generalized scrolling technique could incorporate multiple views of the same large virtual panel. The user could drag rectangles over a large-scale overview actuator to select active areas that would appear in one or more small-scale magnification views.

Three-dimensional control surfaces can strengthen the association of actuator and function. The user can concentrate on the controls required by the current task by changing his or her viewpoint. See figure 4. To build complex, contiguous, three-dimensional control surfaces requires a grouping actuator with a generalized polygonal boundary. This actuator is called Polyframe and is otherwise similar to Viewframe. Conversion of mouse coordinates into the transformed space as discussed above must also be performed.

Multiple selection devices improve the communication bandwidth into the interface. The simplest addition is that of a "shift" key which, in the Panel Library, is used to enter a fine control mode. A second mouse can be used to "hold" or specify a center of rotation while the other mouse is used normally. Taking the idea of multiple input devices further leads to more articulate gesture-based input systems, for example the VPL data-glove and body motion monitors.

9 Combining Actuators and Data

As interface toolkits become more rich, the distinction between the interface and the user's rendered data begins to fade. User data may be rendered within a control panel using the Graphframe

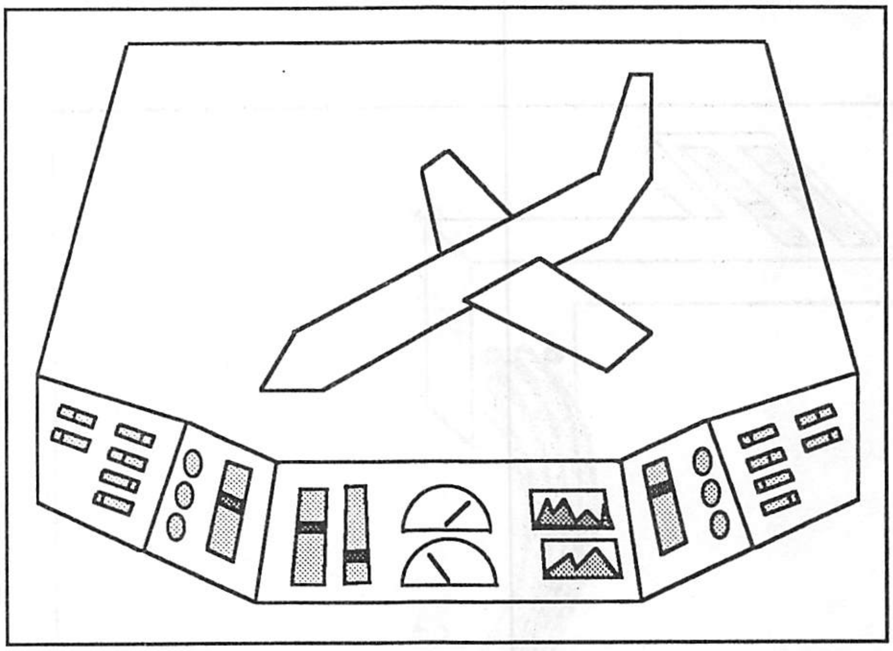


Figure 4: A complex three dimensional control surface associates actuator position and function.

actuator. Similarly, three-dimensional actuators need not be confined to control panels, but may appear alongside rendered data, forming a close intuitive bond between actuator and function. See figure 5. Application interactivity can be further improved by having elements of the rendered scene serve as actuators themselves. Scene elements may return their identity in a selection mode, provide a data value based on where they were selected, or call a function with themselves as an argument.

10 Three-dimensional Tools in a Virtual World

Finally, in a fully visualized three-dimensional virtual world, application control can be enbodied in function-specific virtual tools which have a recognizable three-dimensional visual representation. See figure 6. Selection of application mode, or, in other words, choosing the correct tool will be a challenging problem.

Tools can be selected from an ever-present system menu, tool-rack or "utility belt." A more direct selection could be made using a specific gesture or by voice recognition. Voice selection has application in modifying the behavior of tools as well, saying, for example, "paint-brush," followed by, "smaller brush."

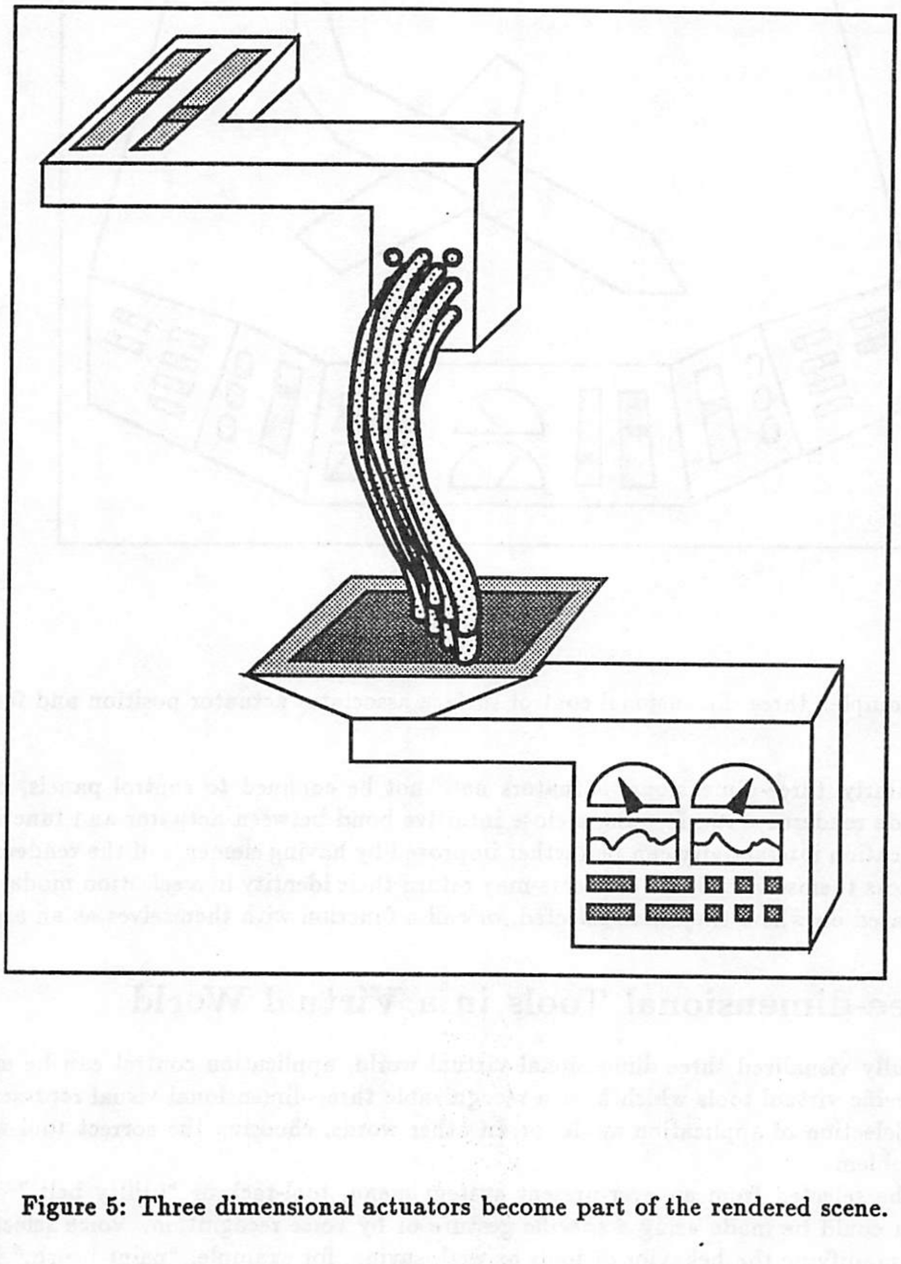


Figure 5: Three dimensional actuators become part of the rendered scene.

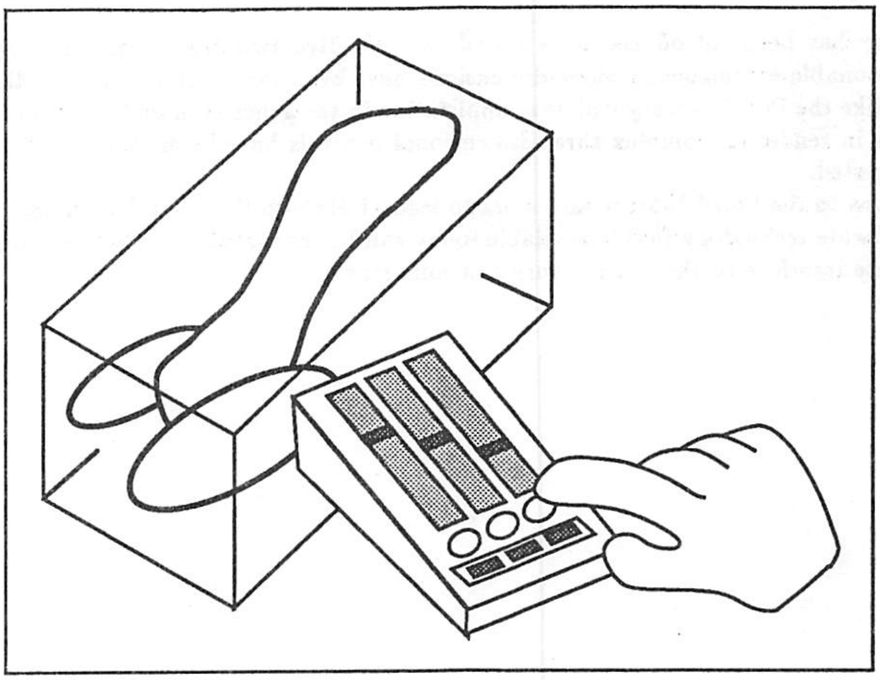


Figure 6: A virtual tool is used to operate on a volume data set.

11 Summary and Conclusions

The Panel Library has been introduced as a useful and effective two-dimensional user interface toolkit. Some reasonable extensions to three-dimensions have been made. It seems clear that user interface toolkits like the Panel Library will have application in the generation and control of virtual worlds. Problems in rendering complex three-dimensional controls have been identified and some solutions are suggested.

These extensions to the Panel Library also seem to lead nicely into the control of virtual worlds. Hardware and software technology that is available today can be incrementally enhanced to provide a rich and complete interface to the virtual worlds of tomorrow.

Learning from a Visualized Garbage Collector

Mark Weiser, Barry Hayes, and Jock Mackinlay
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

1 Introduction

People are beginning to recognize the value of computers as tools for graphical visualization of complex data, tasks, and procedures, particularly for scientific applications [3]. However, relatively few visualization systems have been developed for computer science applications. The exceptions include Model's architecture for distributed visual debugging of knowledge-based systems [5], Myers' presentation techniques for data structures [6], and Brown's algorithm animation system for the teaching of algorithms [2]. Unfortunately, people still seem to be reluctant to develop visualization systems for complex "hardcore" computer science applications, such as compilers, garbage collectors, and operating systems.

We have implemented a visual interface to a garbage collector, ported the interface to a second collector, and used the interface to help us with the ongoing development of these collectors. We think there are three kinds of lessons that we have learned from our experience. First, visualizing can be easy, even for complex computer science problems. The visualization code is roughly 1500 lines of straightforward C code, including code to permit customization of the user interface by the programmer of the collector. The initial version was designed and written in less than a week. The visualizer has proved robust in porting between collectors, and modifications to the collector only rarely need to worry about the visualizer. Second, visualizing is well worth it. We have found bugs and been lead to new insights and algorithms based on observations of the running collector. Finally, the development of effective visualizations requires attention to graphic design principles that are not well known in computer science. We hope that our experience trying to show the many aspects of the running collector has some lessons for others trying to display complex information.

2 Parameters to be Displayed

All of our collectors work by dividing memory into a large number of equal-sized areas called "cards". The size of the cards, which is fixed when the collector is compiled, ranges from about 512 bytes to 8k bytes. Objects larger than one card are allocated

Memory state
- free
- allocated
- unexplored
- visited

Card properties
- A value
- F value
- Unallocated
- Tenured

Figure 1: Garbage Collector Application Parameters

on a sequence of cards adjacent in memory. Objects smaller than a card are allocated by dividing a single card into a number of equal-sized objects, and allocating these objects one at a time. [Memory managers with this kind of allocation are often called "BBOP collectors", an acronym for "big bag of pages."] At any time a particular card is either unused, part of a large object, or divided into small objects.

In a memory management system with a garbage collector, objects pass through a sequence of states (see Figure 1). When they are originally carved out of memory, they are `free`. When the memory manager allocates an object, for example as the result of a call to `malloc`, it must also change that object's state from `free` to `allocated`.

When the garbage collector is invoked, its job is to find objects in state `allocated` which cannot be reached and change their state back to `free`. All of our collectors are modifications of "trace-and-sweep" collectors. In the trace phase of collection, the collector finds all of the objects reachable from the root set, which consists of the machine's registers, the global variables, and the program's stack. It first finds all of the objects immediately reachable from the roots. These objects, whose addresses are found in scanning the roots, are marked as `unexplored`. The collector now visits each `unexplored` object, scans it for pointers to other `allocated` objects, and changes those objects to `unexplored`. After each object is scanned, it is marked `visited`. This continues until all `unexplored` objects have been scanned and marked `visited`.

At the end of the trace phase, all of the objects reachable from the root set have been marked as `visited`, and any object still marked as `allocated` is not reachable. The sweep phase of collection makes one pass over all of the objects, changing `allocated` objects to `free`, and `visited` objects to `allocated`. The space used by the unreachable objects can be reused.

Unfortunately, things aren't as simple as all that. It can be very time consuming to trace all of the allocated memory. Most modern collectors recognize that almost all of the reclaimed storage comes from the storage which was recently allocated and trace only the younger objects. The older objects are treated as roots of the collection, with pointers from older to younger objects discovered by scanning the old objects, and maintaining a cache of these "cross-generational" pointers.

In our collector, cards are assigned to generations based on two values associated with a card (see Figure 1). The card's A value is the time of creation of the oldest object on the card, and the card's F value is the time of creation of the youngest object pointed to by an object on the card.

3 The Design of Our Visualizer

We developed our visualizer with a mixture of principled design and empirical development. The sources of the principles for the design were application-specific knowledge about garbage collection, graphic design knowledge about how to encode application parameters effectively, and a general desire to develop a flexible tool. The reason for the empirical development was our inexperience with the visualization of garbage collection. As we developed the visualizer, we discovered how it could be used, which lead to design improvements. We were also able to test specific techniques suggested by our graphic design principles to see if they actually worked as part of the developing design.

A garbage collector visualizer must present both the components and the component properties associated with garbage collection. In our case, the components of garbage collection form a spatial hierarchy consisting of memory, cards, and objects, and the component properties consisted of memory state and the time values associated with the cards. The design first developed the basic visual representation for the components and then used various graphical techniques to present the properties.

We used the conventional spatial analogy to represent memory, which has increasing heap memory addresses arranged in reading order: left-to-right, and then top-to-bottom. Since memory is large and presentation space is scarce, each row of memory was only one pixel high. Cards were added to this basic arrangement by dividing each row of the visual presentation into a series of line segments representing cards. Objects were presented as individual pixels in the line segments. The result is a grid of line segments representing the cards of memory arranged in reading order by their address.

Given the design of the basic visual representation for the visualizer, the next step was to use various graphic techniques to incorporate the component properties. The French graphic designer, Bertin, identified the following six graphic techniques that are independent of the position of a graphical object: color, size, shape, orientation, texture, and saturation [1]. It turns out that these graphic techniques are appropriate for different types of information [4]. For example, color hue is an effective graphical technique for distinguishing a small number of unordered values, such as the state of the memory. Thus, we use color very effectively in our visualizer to indicate memory state. Color hue is less effective for ordered or quantitative information. Figure 2) lists the graphical techniques that we used to encode our garbage collection parameters.

The A and F values associated with cards are quantitative information. Size is an effective graphical technique for quantitative information, particularly when the graphical objects are aligned for easy comparison of their size. Therefore, we decided to use line segment lengths for the A and F values and position these segments below

Memory state		color hue
Card properties		
	A value	segment length
	F value	segment length and texture
	Unallocated	color hue
	Tenured	saturation

Figure 2: Graphical Encoding of Garbage Collection Parameters

the associated card segments. Empirical development determined that an effective layout was to divide each column of card segments into a column for the A value segments and a column for the F value segments. The result was two narrower columns that could be easily scanned by the eye to pick out A or F values that were relatively large or small. Other quantitative graphical techniques, such as orientation or area size, required too much space to be useful in a garbage collector visualizer.

In addition to these primary graphical techniques, we have used saturation to "gray out" tenured cards and texture to indicate F values that have not been set or are otherwise not valid.

Several parameters of the display are under dynamic user control. At any time, the user can ask for changes in the empty space vertically or horizontally between card segments, in the space between individual objects, and in the number of words of storage per pixel. In separate windows, the user can also adjust the colors associated with each state and control the recording or playback of past collector runs. (Adjusting colors was only important during early exploration, while we were looking for seven pleasing and distinct colors for our different states. But, it was very important then.) The most common parameter for dynamic adjustment is the number of words per pixel, to show more information about individual cards or to fit all of memory onto the screen.

The programming interface makes it easy to visualize an existing card-based collector. Basically, there is a single call for updating the display of an object, which has parameters: object state, length of object, and address of object. Similar calls update the state of a card.

Internally, the programming of the visualizer was extremely simple. For most calls we simply take the object state, look up a color, subtract the object address from the heap beginning, and change the color of the appropriate pixels, according to the object's length. The visualizer has no state, it simply transduces the object by recording calls onto the display.

The programmer of a collector registers a number of procedures with the visualizer, which are connected to buttons on the display. These include one for walking through all storage and calling the object display routines. There is also a generic call for adding new variables to the user-interface to be displayed and modified dynamically via sliders. New sliders can be added at any time during program execution. This permits a mild amount of customization by different programmers.

4 Experience with Visualized Collectors

The visualizer has been installed on two different collectors with similar results. Unfortunately, we did not realize how useful the visualizer would be in the development and maintenance of our garbage collectors and did not make detailed day-to-day observations about our use of the visualizer. Our conclusions about the usefulness of visualized collections is based on anecdotal evidence.

Visualizing garbage collection has proved valuable for many reasons. Debugging the collector with visualized memory is easier than debugging with a standard debugger alone. We could watch the program approach a point where the bug first became evident and often see the cause of the bug before its effects became apparent. One of the more dramatic of these instances lead to a bug being squashed in five minutes with visual observation after several hours of unvisualized debugging.

More importantly, there are many bugs which we discovered only because we had the visualizer. These were usually characterized by a "pattern of suspicious behavior" in the display. Since many complex programs can behave properly but still have bugs, we doubt that these bugs could have been found as easily any other way. Sometimes these patterns would turn out not to be bugs but to be correct behavior. These cases pointed out flaws in our personal models of the behavior of the collector, and led us to have a better understanding of the program.

Finally, visualization is a good instrumentation tool to aid in judging the effects of changes made to the collector. It would be hard to figure out which information to abstract for instrumentation, but the high bandwidth of a color display allows us to dump a vast amount of the data directly to an expert and let a human do the abstraction in a natural way. Being able literally to see ways in which the collector can be improved encourages us to use the visualizer as a general instrumentation tool and write more precise specific instrumentation tools to answer questions raised.

5 Future Work

It seems that, for our current collectors, we have reached a plateau with this style of algorithm visualization. The most valuable information about the collector is displayed, and the display is meaningful to people familiar with the collection algorithm. While we may find additional information to display continuously, any information added to the display risks making it harder to read what is already displayed. Reuse of graphical presentation techniques can interact with their previous use, and we are beginning to run out of independent graphical techniques that do not require additional space.

The most likely avenue for development is to use the display as a specialized visual debugging tool. Detailed views of parts of memory, including actual values of memory locations, could be brought up by pointing and selecting cards. From that, tracing pointers could be done by hand. We could enhance the input quite freely, letting a developer create objects through the display panel, link them together, and then watch the collector work on the data created.

6 Conclusions

Having a visual output from the garbage collector greatly enhanced our ability to debug it and gave us better mental models of the behavior of the program. The visualizer was easy to implement, did not hinder the development of the garbage collector, was particularly useful during the initial development of the collector, and continues to be useful even today. Our experience suggests that the development of visualizers for hardcore computer science application is well worth the effort.

References

[1] Bertin, J. (1983) *Semiology of Graphics*, W. J. Berg, Tr. University of Wisconsin Press, Milwaukee, Wis.

[2] Brown, M. H. (1988) *Algorithm Animation* MIT Press, Cambridge, MA.

[3] Special Issue on Visualization in Scientific Computing *Computer Graphics* Vol 21(6, Nov.), 1987.

[4] Mackinlay, J. (1986) Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics,5*(2, April), 110-141.

[5] Model, M. L. (1979) *Monitoring System Behavior In a Complex Computational Environment* PhD Dissertation Stanford University. See also Technical Report CSL-79-1 Xerox PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304

[6] Myers, B. A. (1980) *Displaying Data Structures for Interactive Debugging* Technical Report CSL-80-7 Xerox PARC 3333 Coyote Hill Road, Palo Alto, CA 94304

The Render Button:

How a High-end Graphics Company Developed a Personal Graphics Product

Jon H. Pittman
Director of Software Development
Wavefront Technologies, Inc.
530 E. Montecito St.
Santa Barbara, CA 93103
(805) 962-8117

[decvax!]ucbvax!ucsbcsl!wavefro!jon

Abstract

The Personal Visualizer™ was developed to enable a casual user to create high-quality rendered images. Providing sophisticated rendering techniques to casual users forced us to confront a number of issues and to undergo a shift in values. These issues are outlined, and the ways we dealt with them are discussed. Things we would do differently as a result of our development experiences are also discussed.

Introduction

In July of 1989, Wavefront Technologies, Inc. introduced a product called the Wavefront Personal Visualizer™. Until that time, Wavefront had been known for high-end rendering and animation software targeted primarily toward entertainment and creative graphics. In other words, our products were directed toward people who used expensive graphics workstations to make "pretty pictures" for a living. The Personal Visualizer is oriented toward casual users who want to produce realistic images on low-cost engineering workstations. These people produce images as a means to an end rather than as an end in itself. They are typically engineers or scientists who use images as one tool in a set of tools to enable them to do their primary task. They use the product one or two hours a week rather than forty to eighty hours per week.

Developing the Personal Visualizer resulted in quite a culture shift at Wavefront. Where previously we focused on achieving ever-higher levels of photorealism and realistic motion, the Personal Visualizer forced us to learn about usability, information structuring, graphic user interface design, and simplicity. We had to expand our point of view beyond features and performance to make our product effective for the casual user. We found that the overriding design issue for our personal visualization product was one of conceptual integrity. To make visualization accessible for a casual user, we had to develop a clean, coherent product in which all of the components related logically to each other and acted as an integrated whole. We found the words of Fred Brooks, author of ***The Mythical Man Month*** [1], particularly apropos to our development effort:

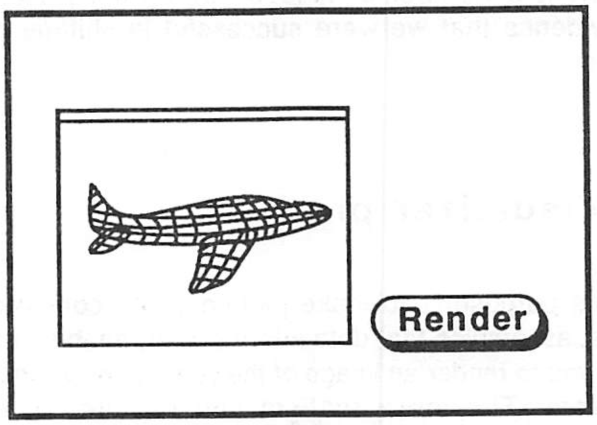
> *" ... conceptual integrity is **the** most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one which contains many good, but independent and uncoordinated ideas."*

For an organization such as ours, developing a personal visualization product was often a process of determining which features to omit in an attempt to achieve conceptual integrity. This was quite a challenge for an organization used to building products for expert users. In this paper, I will describe the Personal Visualizer project and product. I will then discuss some of the specific issues we confronted in developing the Personal Visualizer. While discussing these issues, I will speculate on some of the things we would do differently as a result of our experiences and on some of the directions we can expect to see personal visualization products take in the future.

The Personal Visualizer project

We originally began discussions with a graphics workstation vendor in January of 1988 on a rendering project. They wanted to take advantage of Wavefront's software rendering capability to make wireframe and hardware rendered images look "pretty." The request was for a magic "render button" that, when pressed, would turn a wireframe or hardware shaded image into a beautiful photorealistic image. When originally approached, we were somewhat skeptical of this endeavor. "Surely," we were told, "it can't be that difficult to add a button to the screen to render an image!" Of course, the request was somewhat naive. The problem was not one of adding a button to make the wireframe beautiful. It was one of underlying data representation and manipulation of all of the elements necessary to make a photorealistic picture.

Currently, making a photorealistic picture or animation is very much like making a Hollywood film. Before creating the picture, you must carefully construct a scene which contains all of the elements in the picture. In addition to constructing the spaces and objects in the scene, you must carefully place lights to establish the mood or atmosphere, place cameras to establish the view, apply surfaces to the objects to give them texture and character, and place backdrops to establish an environmental context. Of course, producing an image with more realism requires more detail and accuracy in the data describing the scene. To further complicate things, many of the techniques used are not direct analogues to the real world but tricks of the "smoke and mirrors" variety that are used simply to produce the picture. They are similar to stage sets which provide the illusion of reality when seen from a particular viewpoint. In the case of the workstation vendor who wanted the render button, most of the wireframe and shaded images on their graphic workstations were of CAD or scientific data. The underlying data structures did not have all of the scene information necessary to construct a realistic image. They contained only enough information for simple display techniques. They did not contain information rich enough to construct a photorealistic image.



A render button to make the wireframe image photorealistic

The Personal Visualizer was developed to fulfill the promise of the render button. It is designed to enable a casual user to create a photorealistic image. The Personal Visualizer provides a data management system to assemble into a ***scene*** all of the components necessary to make a picture. Once the scene is described, you may generate a picture or sequence of pictures at various levels of realism. You need not create geometric data; data translators allow geometric data to be imported from a CAD system or modeling package.

The Personal Visualizer is organized into a core product that provides basic scene assembly, data management, and rendering capabilities; and a set of options that extend the functionality of the core. The options provide capabilities to create geometry, motion, or surfaces; output images to video; and more extensive import of geometric data from CAD systems. The core Personal Visualizer is bundled on graphic workstations and the options are sold by Wavefront. The options allow a user to configure a visualization tool to meet his or her specific needs.

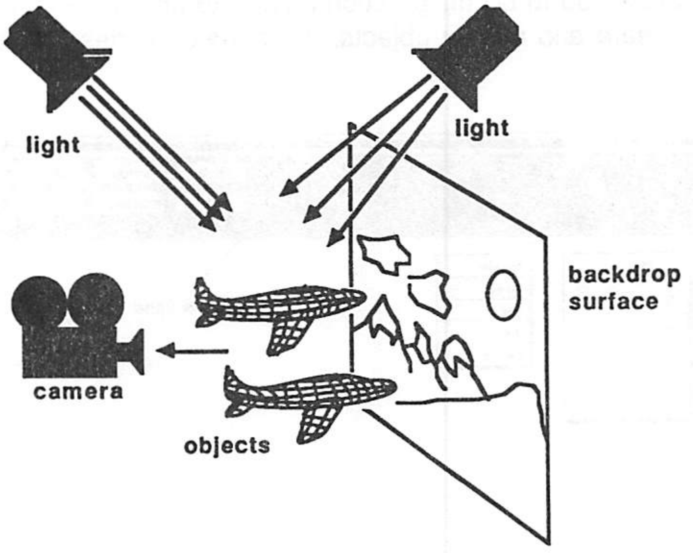
The user interface was the primary focus of the Personal Visualizer development effort. We developed it around a graphic user interface to provide access to a wide range of users. Since the Personal Visualizer was intended for a broad audience, we tried to create a simple, concise product that was easy to understand and use instead of producing a feature-laden application. Whenever we were confronted with a choice between a powerful but complex feature and reduced but conceptually clear functionality, we chose conceptual clarity and integrity.

This focus on usability and conceptual integrity was difficult for many members of the Wavefront staff. We were accustomed to pushing the envelope of rendering and animation technology. Our customers were, by and large, interested in more and more sophisticated rendering and animation features. Our previous focus was on technology rather than designing for conceptual clarity. Our culture was one which was very typically UNIX-oriented in that we viewed products as sets of small, useful tools which were strung together to achieve a desired effect or result. Trying to build a product for the casual user forced us to confront a number of issues differently. However, the Personal Visualizer product is evidence that we were successful in shifting our focus from the expert to the casual user.

The Personal Visualizer product

The Personal Visualizer's purpose is to make pictures. The core product collects data from a variety of sources, assembles that data into a scene, enables you to arrange lights and cameras, and allows you to render an image of the scene. The scene can be rendered at various levels of realism. The more realism you require, of course, the more information you have to add to the scene and the more time the image will take to render. The information you assemble to form a scene consists of a set of resources. The Personal Visualizer works with the following kinds of resources:

- **Objects** are geometric data. They provide the shapes that you will manipulate and place in space to create an image. Objects may be as simple as cubes or spheres or as complex as airplanes or buildings. Objects may be grouped together to form more complex objects.
- **Surfaces** describe color, texture, reflectance, and transparency properties. Surfaces are applied to objects much like decals are applied to a model airplane. The surface that is applied to an object will dictate the level of realism of the rendered image. Alternatively, surfaces may be used as backdrops.
- **Lights** describe the direction, color, and falloff of the light sources in a scene. The lights help establish the mood of the scene and interact with the surfaces to produce the rendered images.
- **Cameras** describe the view of the scene. The actual camera resource describes attributes such as focal length. The position and direction of the camera dictates the actual view of the scene.
- **Pictures** are the end results of the rendering process. You can render a scene and save it as a picture.



A set of resources comprising a scene

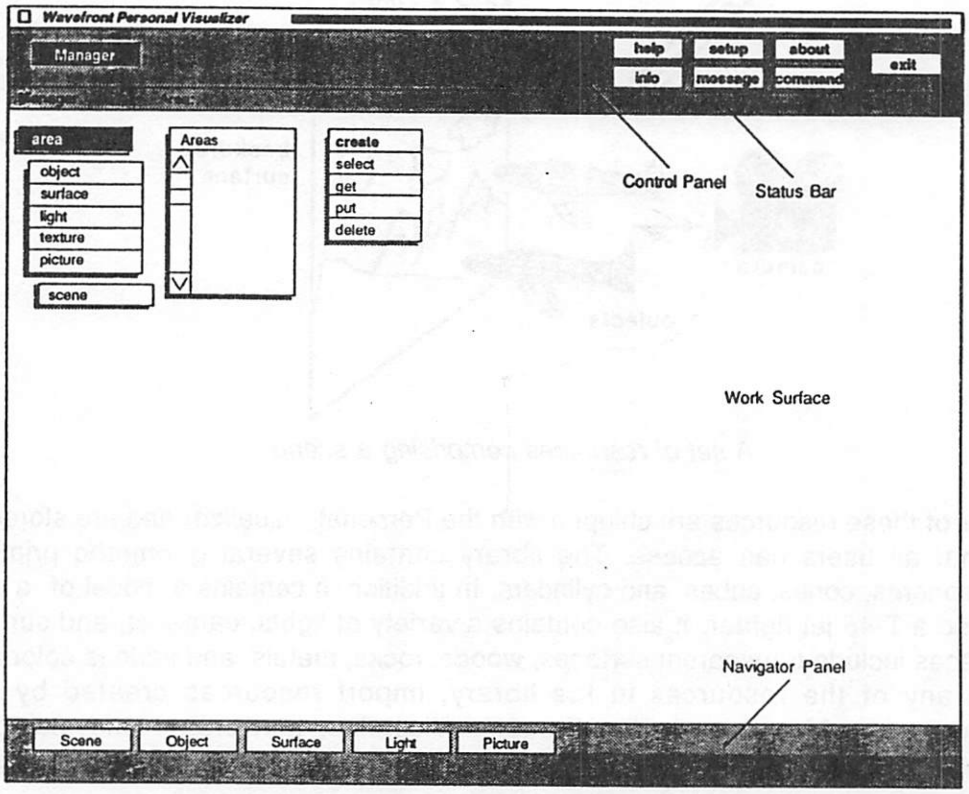
A number of these resources are shipped with the Personal Visualizer and are stored in a library that all users can access. The library contains several geometric primitives such as spheres, cones, cubes, and cylinders. In addition, it contains a model of a space shuttle and a T-45 jet fighter. It also contains a variety of lights, cameras, and surfaces. The surfaces include transparent surfaces, woods, rocks, metals, and various colors. You can use any of the resources in the library, import resources created by other applications, or add options to the Personal Visualizer that create and manipulate resources. The figure on the previous page shows an example scene with various types of resources.

The "face" of the Personal Visualizer consists of several different screens or ***editors***. Each editor is focused on a particular task that the user performs. In the core Personal Visualizer, there are two editors:

- **The Manager** is the editor that you enter when you start the Personal Visualizer. It helps you organize and manage resources in your own private work areas. It is where data from other sources is imported to your work areas and data already in your work areas is exported for use by other programs or tools. The Manager also allows you to copy resources from the standard library and to share resources with others through a mechanism called the exchange.

- **The View Editor** is where you assemble scenes and render images. It allows you to add objects and lights to a scene. You can manipulate each of these resources using standard graphics transforms (translate, rotate, scale). You can also point lights or cameras at objects or associate surfaces with objects. Once a scene is assembled, you can test render an image to the screen or render a picture to save it for future use.

Optional editors that allow you to do things such as create and modify surfaces, paint on or annotate pictures, create and modify objects, or animate scenes can be added to the core product.



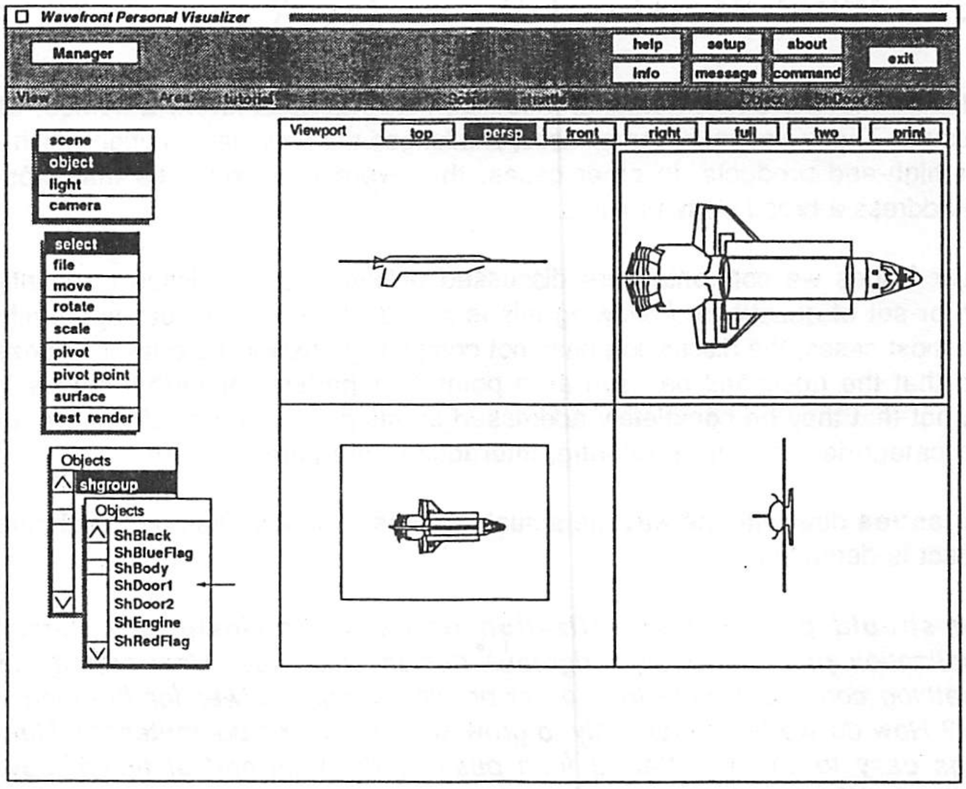
A visual framework common to all editors provides an identity for the Personal Visualizer

Although each editor has a unique function, all of the editors share a common look and feel. A set of elements appear in every editor and form a visual framework for the system. This framework establishes an identity or personality for the product. The elements make up this framework as shown in the figure above:

- **The control panel** -- allows you to get information about the state of the Personal Visualizer, modify preferences, invoke the command language, return to the Manager, or exit from the Personal Visualizer.

- **The status bar** -- indicates which items on the screen are currently selected.

- **The work surface** -- is the background area of the screen. It is the place where menus, dialog boxes, viewports, and other graphic elements appear.



The Personal Visualizer user interface is comprised of common graphic elements that form a consistent graphic vocabulary.

In addition to the common elements that appear in every editor, each editor is constructed from a standard set of graphic elements. These elements include menus, viewports, dialog boxes, and property sheets. The figure on the following page shows examples of some of these elements. We used these elements throughout the Personal Visualizer to ensure a consistent visual vocabulary and style. We tried to achieve conceptual integrity that was as effective as that of the Xerox Star™ [2,3] and the Apple Macintosh™ desktop.

To supplement the graphic user interface, we incorporated a programmable command language that provides access to all of the functionality of the product. This allows experienced users to work more quickly and to construct "programs" that provide higher-level functionality tailored to a particular use. We crafted the command language with the same view toward consistency and conceptual integrity that we used for the graphic user interface.

Issues - What decisions did we face?

In developing the Wavefront Personal Visualizer, we had to confront a number of tough design issues. These issues, in some cases, challenged the basic tenets upon which we had built our high-end products. In other cases, they were new concerns that arose from trying to address a broad class of users.

The major issues we confronted are discussed below. First, the issue is defined as a question or set of questions. Following this is a brief discussion of our approach to the issue. In most cases, the discussion does not completely answer the questions posed. It is intended that the questions be used as a point of departure for further discussion and thought, not that they be completely addressed in this paper. These issues are arranged into four categories: design, rendering, interaction, and infrastructure.

Design Issues deal with the way the actual features, look and feel, and configuration of the product is determined.

- ***How should personal visualization products be designed?*** *Personal visualization products pose a rigorous design challenge. How do we make something complex simple to use yet provide enough power for the high-end user? How do we layer capability to grow with the user's competence? Making things easy to use is different from pushing the high-end of technology. It requires different skills than devising a ray-tracing algorithm or radiosity technique. What skills and approaches to product design are required for personal visualization?*

  In designing the Personal Visualizer, we used a full-time product designer dedicated to the project. His job was to ensure that the conceptual integrity of the product was maintained. He worked extensively with the developers, technical writers, and marketing staff to ensure that the product was perceived as an integrated whole. Although the design effort was successful, in retrospect we would do some things differently:

  * **Build a user's conceptual model early in the process and maintain it.** We should have built an explicit model of how our user perceived the product and constantly tested the design against that model.

  * **Build and maintain a glossary.** We often found over the course of the project that we called something by many different names or, conversely, that we overloaded certain terms. We should have built a glossary of all of the terminology that the user saw and constantly evaluated the glossary against the user's conceptual model.

  * **Prototype.** We spent a lot of time discussing various design alternatives. However, most people could not evaluate and critique design decisions effectively until they became concrete through

representation in a prototype. We did most of our prototyping on paper. We should have developed more interactive prototypes and made rapid prototypes an integral part of our design and development process.

* **Usability test.** When the product got into the hands of real users, we often found that they had problems we did not anticipate or had good ideas that were too late to implement. We would do much more usability testing with real users early on in the design and development process.

- ***How do you build the correct user's conceptual model?*** *Who are the users of a personal visualization product and what characteristics do they have? Users may have a broad range of skills and computer and graphics literacy. How do you make a personal visualization product easy for the novice yet powerful enough to satisfy the proficient? Is it appropriate to try building a product that attempts to serve the needs of both the expert and the novice user? What is the user's mental model of the tool and how can it be reinforced by the product?*

  The best way to build a model of how the user views the system is to understand your user thoroughly and frequently test the model against the user and against the product. We had difficulty understanding our users, since most of Wavefront's traditional user base consisted of professional animators. It was difficult for them to place themselves into the shoes of a novice user. We got the most valuable information from users once we had the product out into their hands in the testing cycle.

- ***Is it better to provide one universal tool or lots of little tools?*** *To provide seamless access to visualization tools, two choices exist. You can create a "universal system" which does many things, or a collection of small, focused tools that talk to each other. Which is more effective?*

  To the user, this should not make a lot of difference. The structuring of applications into tools or editors is more an artifact of the way we design and implement user interfaces than the way someone would like to work. In our case, we provide a universal tool in that we provide a framework which is consistent throughout the Personal Visualizer, but we break functionality into separate editors that have limited scope and function. In effect, our editors are viewed as small, focused tools by the user. This kind of partitioning of functionality seems to be effective in keeping each tool simple. However, as the number of tools grows, it is difficult to keep navigation between the tools simple and to provide a clear picture of how total functionality is segmented between the tools.

- ***How do you avoid creeping featuritis?*** *Personal visualization is a new, exciting technology. Everyone has features and functions they would like to see incorporated in personal visualization products. The temptation is to become all things to all people. Unfortunately, this may result in a product that does a lot of* *things but none of them well. How do you develop a visualization tool with broad appeal, yet maintain a clear, clean, simple product?*

Time and market constraints help with this problem. We found it a constant battle to keep the product from becoming an accretion of interesting but unrelated ideas and features. We had to constantly focus on maintaining simplicity and integrity. Of course, as the end of development loomed closer and people realized that infinite time to develop was not available, it was much easier to find out what was important and what was not. In addition to time pressure, maintaining a clear focus on the product's purpose and intended user can help control random feature addition and maintain conceptual integrity.

Rendering Issues deal with the way a personal visualization product deals with graphic display and rendering.

- ***What level of realism is appropriate?** In the entertainment market we assumed photorealism was the goal. However, to achieve photorealistic effects, we often have to go to great pains to set up data. Are users willing to pay the costs of photorealism? Can we make it easy for "Joe Public" to achieve realism? Does "Joe Public" want to achieve photorealism? Does photorealism obscure or enhance the message? Are there ethical questions concerned with producing photorealistic images?*

  We provide a range of levels of realism in the Personal Visualizer. In fact, we make extensive use of hardware shading, and the user can interact with a scene using a very high level of realism without resorting to software rendering. It is clear that many users do not need or want a high level of realism. However, we feel that the standard of realism that users expect to see will rise as the capability to produce such realism becomes accessible and the cost becomes reasonable. We must keep in mind, however, that most people who use computer graphics are trying to do useful work, not just produce pictures. Pictures enable them to do something. Realism may or may not help that person get their job done. It should be viewed simply as another tool in their repertoire.

- ***What happens when graphics become ubiquitous?** Workstation vendors are all interested in providing graphics capability. Hardware shading and intrinsic graphics libraries are becoming standard on many workstation products. How does this affect the development of personal visualization products? Will this cause a shift of focus from rendering techniques to interaction techniques?*

  We designed the Personal Visualizer to take advantage of hardware rendering capabilities that exist on various graphic workstations. It is structured such that the transition from hardware rendering to software rendering is seamless, although there is a discontinuity from interactive performance to waiting for a picture to render. It is clear that more and more rendering capabilities will be built into hardware or into low-level system software. Software techniques, however, will be able to provide more realism for some time to come. The real problem, though, is not one of rendering or realism. The real problem is how to use sophisticated rendering technology that comes with the hardware. Harnessing all of the "gee-whiz" rendering techniques to do something useful is a much different problem from rendering. Doing useful work with hardware rendering requires good interaction techniques. When realistic rendering becomes ubiquitous, we will be able to turn our attention to the real problem at hand, how to use visualization techniques to accomplish useful work.

Interaction Issues deal with the dialog between the human user and the personal visualization product.

- ***What style of interaction is appropriate?*** *Two traditional alternatives are graphic interaction and command-based interaction. Some people prefer to use typed commands or scripting languages to interact with an application. Others prefer graphic user interfaces. Which is more appropriate? Are the two approaches mutually exclusive?*

  Although the current trend is toward graphic user interfaces, there are still a lot of reasons to use commands that you type at the keyboard. Commands can be more succinct and precise than graphic interactions, can be grouped into scripts for repeated execution, and may be combined with programming constructs to allow extension of the base functionality of the product. We are starting to see scripting languages incorporated into graphic user interfaces. HyperTalk, HyperCard's scripting language, is a good example of this. Combining a graphic user interface and a programmable command language results in more power and flexibility than either approach taken separately. Providing both allows users to select the interaction mode that best suits their needs. We provide both in the Personal Visualizer. In our implementation, the command language and graphic user interface are relatively independent of each other. You have to switch back and forth between the two styles of interaction. A superior implementation would closely integrate the two means of interaction so they worked together.

- ***How should a personal visualization product deal with multimedia?*** *Media such as sound, video, kinesthesia, and tactile are becoming more pervasive. Soon they will be integrated into hardware and software environments. How can a personal visualization product exploit these capabilities? For example, now video is expensive and messy. How can we make it easy and inexpensive for the user to access?*

  In the Personal Visualizer, we used one medium (the graphic display), three channels (mouse and keyboard in and graphic display out), and three dimensions. However, this is only a modest taste of the possibilities in a personal visualization tool. There is no question that personal visualization tools should be integrated with multimedia technology. We interact with the real world using all of our senses. It is imperative that we interact with our visualization tools in a similar fashion. It will be necessary to build mechanisms into the infrastructure needed for personal visualization tools to support multimedia, multi-channel, and multi-dimensional interaction between the user and computer. This will substantially increase the communication bandwidth between the user and the computer. Currently, UNIX workstations don't even deal well with video output, let alone other types of media. Thus, we still have a long way to go before dealing effectively with multimedia.

Infrastructure issues deal with the ways personal visualization tools relate to their operating environment (hardware, UNIX, graphics libraries) and other visualization tools.

- ***What is the role of UNIX?*** *UNIX has many features that allow clean integration of tools and provide power to the expert user. However, UNIX can be quite daunting to the casual user. How can a personal visualization product coexist peacefully with UNIX? Should the two ignore each other or is there a graceful way to integrate the best of both worlds?*

  Our charter for the Personal Visualizer was to make visualization as accessible as possible for the casual user. One of the most difficult learning problems for new Wavefront users is learning about and dealing with UNIX. In addition, there is currently a lot of work in the UNIX community aimed at hiding the raw UNIX user interface and providing a more accessible interface such as Silicon Graphics' WorkSpace [4], NextStep, and OpenLook to make UNIX appealing to a broader range of users. Thus, we chose to ignore UNIX once a user has invoked the Personal Visualizer. As the work on making UNIX more accessible progresses, we expect to tie the Personal Visualizer more closely to standard UNIX graphic user interface techniques.

- ***Where does the data go?*** *Graphics applications require lots of data and various data elements may be closely related. The UNIX file system and a byte-stream data model may not be appropriate for a personal visualization product. The complexity of graphic data may not lend itself to such an approach. How do we organize data and present that organization to a casual user in a way that makes sense? How can we make the management of data seamless and effortless for the casual user?*

  A scene may contain a lot of data and there may be complex relationships between data elements. If we kept all the data for each scene in one file, the file would be big and cumbersome and it would be difficult to share data elements between scenes. If we kept the scene data in a lot of small files, a user might inadvertently corrupt the scene by removing or renaming some of the files. Ideally we wanted a file system that let us control user access. Since we did not have such a scheme available, we decided to hide the database from the user. The database exists in a hidden subdirectory below the user's home directory. We provide data manipulation capability in the manager. The user is discouraged from directly manipulating the database. This was (and still is) a fairly controversial issue within Wavefront. The other alternative is to let the user use the UNIX file system directly. Given our approach of hiding UNIX from the user, letting the user deal with the file system directly did not seem appropriate.

- ***How do we fit personal visualization tools together?*** *To allow seamless integration of tools, a number of standards are needed. These include data interchange, look and feel, portability, networking, live links, and peripheral device standards.*

  UNIX has a tradition that supports building a lot of little, focused tools, and stringing them together to do useful work. The operating environment supports such a model through the file system and mechanisms like pipes. It works very well for byte-stream data such as text. It does not work so well for complex graphic data. The intent behind UNIX is a good one. The models it supports, however, are not appropriate for the kinds of data that must be supported in a personal graphics system. Supporting personal graphics requires a new infrastructure that deals with graphic data and the interaction of graphic software components. Lacking such an infrastructure, we had to build our own. I have to admit that, although the intent was to build an infrastructure that supported the linking together of a lot of small tools, the one we developed was a bit too monolithic. We are now working to rectify that problem.

- ***How does a personal visualization product adapt to changing environments?*** *The hardware and operating environments in which personal visualization products operate is changing very rapidly. New hardware, applications, peripheral devices, communications technology, etc. are continually evolving. How can a personal visualization product adapt to these changes?*

  Eventually, the underlying hardware and operating environment will be of little concern to the developers of visualization tools. We are already beginning to see less reliance on hardware environments because of the proliferation of UNIX. This is the result of the use of standard hardware components and the standardization of the UNIX operating system. Unfortunately, no clearly established standards exist for graphic user interfaces, graphics display libraries, or rendering. As these standards become established, visualization tools will adapt to new environments easily. In effect, they will be tied to an infrastructure which abstracts them from the details of hardware and operating environment.

One common theme throughout many of these issues is that we need an infrastructure to support personal visualization tools. In the absence of such an infrastructure, we had to invent one for the Personal Visualizer. However, it was a massive effort. It is not practical for everyone who wants to build a personal visualization tool to build such an infrastructure. We needed something that would abstract hardware, peripheral devices, operating environment, and graphics display and interaction in such a way that anomalies are avoided. UNIX was designed as such an infrastructure for text-based systems and has proven to be a very robust infrastructure for such systems. However, it is beginning to show signs of stress when dealing with personal graphics systems. What is needed is a new infrastructure (or extensions of the old infrastructure) that supports the needs of personal visualization tools.

Conclusion - What does the future hold for personal visualization?

The Wavefront Personal Visualizer is the first step toward personal visualization tools that will be commonplace in the future. The real limitation of the Personal Visualizer is that it is primarily an output tool. It shows the results of rendering a scene that represents some real-world or artificial environment or phenomenon. It has limited capabilities to manipulate, interact with, and explore that environment or phenomenon. We have a long way to go before we fully achieve the kind of pictorial conversation discussed by Rob Myers in 1986 where a true graphic dialog is carried out between humans and computers [5].

I hope that personal visualization tools of the future are truly transparent to the user. Graphics will be ubiquitous. Just as the text-based computer now permeates our lives so, too, will graphics-based computers. They will deal inherently with three dimensions since the world that they interact with and represent is three-dimensional. They will also be multimedia and will carry on dialog with humans through multiple communication channels. We will need to build visualization tools that can harness all of this communications and capability.

Although the graphical user interface of the Personal Visualizer is a big step toward making photorealistic rendering accessible to the casual user, it is my hope that we will look back on such tools as complex and cumbersome in the future. We need to strive for personal visualization tools that enable us to manipulate objects and environments in the computer as easily as we manipulate them in the real world.

Acknowledgements

The Wavefront Personal Visualizer was developed by Chris Kitrick, Dave Immel, Julie Daily, Cliff Brett, Lindy Lindstrom, Jay Tomlinson, and Josh Aller. Roy Hall, Kim Shelley, Don Brittain, and Jon Barber developed software components that were used in the Personal Visualizer. Derry Frost was the product designer and was responsible for the conceptual integrity of the product. Diane Ramey, Catherine Babine, and Brad Weed wrote the documentation and the on-line help. Steve Miley was responsible for configuration management and release engineering. Jim Hill and Patsy Crawford tested the product and helped customers with beta testing. Karen Gadway provided administrative support. Each of these individuals as well as many others contributed to the success of the project and grappled with the issues described in this paper.

The Wavefront Personal Visualizer was developed in conjunction with Silicon Graphics, Inc. Silicon Graphics provided the original impetus for the project and was the originator of the "render button" concept. Tim Heidmann, Ed Bank, and Trish Jones from Silicon Graphics worked closely with us in our development effort.

Julie Daily, Derry Frost, Diane Ramey, Chris Kitrick, Dave Immel, Karen Gadway, and Care Heller-Pittman made helpful suggestions and comments in reviewing this paper for publication. Derry Frost and Diane Ramey assisted in preparing the figures.

References

[1] Frederick P. Brooks, *The Mythical Man-Month, Essays on Software Engineering*. Addison-Wesley. 1975. Page43.

[2] Dr. Daniel E. Lipkie, Steven R. Evans, John K. Newlin, Robert L. Weissman. Star Graphics: An Object-Oriented Implementation. *Computer Graphics*. Vol 16., Number 3. ACM Siggraph. July 1982. Page 115-124.

[3] Jeff Johnson, Theresa L. Roberts, William Verplank, David C. Smith, Charles H. Irby, Marian Beard, Kevin Mackey. The Xerox Star: A Retrospective. *Computer*. Vol 22. Number 9. IEEE Computer Society. September 1989. Page 11-29.

[4] *Working in a New Space*. Silicon Graphics, Inc. 1989.

[5] Rob Myers, Pictorial Conversation: Design Considerations for Interactive Graphical Media. *Proceedings of the 1986 Monterey Computer Graphics Workshop*. USENIX Association. November 1986. Page 17-35.

Part-Task Flight Simulation on a UNIX Graphics Workstation

Steven H. Philipson
Sterling Federal Systems, Inc.
1121 San Antonio Road
Palo Alto, CA 94303
(415) 964-9900 ext. 218
ARPAnet: steve@eos.arc.nasa.gov

Stefan J. Jeffers
International Business Machines
Advanced Workstation Division
1510 Page Mill Road
Palo Alto, CA 94304
(415) 855-4431

Abstract

Aviation safety researchers at the NASA Ames Research Center have been investigating advanced cockpit technology and flight displays on large scale, full mission capable flight simulators for several years. These simulators are driven by large multiple mainframe computer systems. Recent advances in computer workstation technology have allowed software for flight simulation and instrument display subsystems to be quickly developed and run on single-user workstations, thus providing research opportunities at a fraction of the cost and time of that for full scale simulation. This presentation discusses the development of a part task flight simulation and instrument display system on a Silicon Graphics, Inc. IRIS 4D UNIX computer graphics workstation.

This work was performed by the authors as employees of Sterling Federal Systems, Inc., in contract support of the NASA Ames Research Center, Moffet Field, California. The project manager and principal investigator was Alfred T. Lee, PhD, of the Aerospace Human Factors Research Division (ASHFRD) of NASA Ames.

Introduction

NASA Ames Research Center is engaged in the study of human performance as it relates to flight safety in the operation of jet airliners. One research approach is to observe flight crews as they operate aircraft in both real world and simulated flight environments. Ames employs several large scale flight simulators in this work including a motion-base Boeing 727 simulator and a fixed-base Advanced Concepts Flight Simulator (ACFS). The simulators are located at the Man-Vehicle Systems Research Facility (MVSRF).

The 727 simulator serves as a research platform for operation of mature technology aircraft. The ACFS is used in the study of state-of-the-art and next-generation systems, including crew member interaction with computer driven electronic flight instrumentation systems (EFIS). EFIS displays are used for both primary aircraft control tasks and a variety of secondary tasks. These secondary tasks include control and monitoring of aircraft systems, engines, fuel state, interactive display of weather products and communication with Air Traffic Control (ATC).

The simulators are designed to provide a highly realistic flying environment that closely approximates the real world. Typical "flights" include simulated interactions with ATC, airline company dispatchers, other aircraft, and weather systems, all while the aircraft itself must be correctly operated. The complexity of these simulations requires the support of numerous personnel, several large scale mainframe computers and numerous smaller dedicated-function machines.

The various parts of the simulators interact closely, thus any new elements must be carefully integrated in order for the entire system to function properly. Display hardware is optimized for real time performance and not for ease of software development. When available, development tools are complex and generally not easy to use. In addition, development and integration require exclusive access to the simulator. During such time it is unavailable for research flights. Development is thus limited to a small number of hours per day.

All of the above factors combine to slow the pace of development of new capabilities and contribute to high costs, thus limiting the number of experimental systems that can be created and tested.

One method of circumventing these difficulties is to study subsystems independently. Individual operational tasks or small groups of tasks can be separated out from the full simulator environment for independent development and testing. Simulation of these restricted domain task sets is referred to as part-task simulation.

The part-task simulation approach requires the availability of machines that are capable of handling all of the elements of the part-task on their own. Until recently, part-task research on human interaction with computer driven electronic flight displays was hampered by the lack of availability of machines that could generate displays of acceptable quality at a high enough rate and reasonable cost. The emergence of high-power computer graphics workstations that possess both powerful display and compute capabilities has enabled development of part-task simulation on a single machine. Several of these new workstations incorporate the UNIX operating system and software development tools which together greatly facilitate the development of new systems.

This part-task simulation does not represent new technology. It was an attempt to create a research testbed based on known simulation methods. This approach promised to yield increased flexibility and adaptability while simultaneously requiring a smaller investment than with earlier systems. The project was undertaken with the expectation that the use of such workstations would allow the testing of a greater number of new display formats and concepts, and that this would promote more effective use of the large scale flight simulators.

Hardware

Several vendors produce workstations that include high-resolution computer graphics displays. Few of them have the performance required for this type of application. A display update rate of 30 Hz was desired, with 15 to 20 Hz being the minimum acceptable rate. (In this context, update rate refers to the number of frames that can be drawn per second and not the scan refresh rate of the display monitors.) A high resolution display (1024 x 1024 pixels, minimum) was desired to enable high quality rendering of aircraft instruments and to minimize "jaggies" or aliasing artifacts. The displays would have to be driven by aircraft flight simulation software, hence substantial general purpose computing power was required to allow performance of these operations in real time. Display complexity was expected to increase over time, so it was desired to acquire a system that would have an upgrade path for increased graphic and computational throughput.

The Silicon Graphics 4D-60 system was selected as an initial development system. At the time of selection (August, 1987), this machine was very competitive in the marketplace in terms of cost / performance ratio. Its specifications indicated that it would be able to handle our initial compute and graphics requirements, and an upgrade path was already in place -- a CPU enhancement was available and a graphics enhancement was in development. Another factor in its favor was that the company had a history of producing upwardly compatible product-line enhancements. Finally, the Ames Research Center had an established base of SGI equipment and in-house expertise which could serve as a support base, thus lowering technical risk.

The capabilities of the 4D series workstation have grown substantially over time. Improvements in compute and graphics display power have kept pace with our increasing requirements as our software has grown in complexity. We found that we were able to maintain (and actually increase) the frame refresh rate without optimizing our code simply by purchasing processor upgrades over time. This allowed us to focus our efforts on developing new capabilities and freed us from the need to increase the performance of existing code. The initial 4D-60 G system was first upgraded to a 4D-70 G and has recently been upgraded to a 4D-80 GT. Further increases in computational power are already available through the SGI Power Series workstations.

Software Development

Through a market survey we investigated in detail two commercially available software products that could produce flight instrument displays. These were VAPS, produced by Virtual

Prototypes Inc., and the Generic Visual System (GVS), produced by Gemini Technology Corporation. (TIGERS, a product of CAE Electronics limited, was not available when this project began.) These packages produce high quality display images but did not match the project requirements. Thus a decision was made to begin development of software in-house that could be tailored to suit research objectives.

The original goal of this project was to evaluate new types of severe weather information and warning displays for in-flight use by flight crew members. Initial efforts were concentrated on creating static displays for evaluation by experienced airline crew members. Such displays were created through the use of a simple paint program and a graphics symbol editor written for this application.

It was desired that the experimental displays appear in the context of an aircraft instrument panel, thus there was a requirement to place primary flight instrument displays on the screen adjacent to the experimental displays. We were relatively inexperienced with the SGI machines, so we simply tried to write some "quick and dirty" code to provide static displays of these instruments. We were able to generate these displays successfully, and subsequently found that the high performance of the workstation allowed this code to be called repetitively with refresh rates of approximately 20 Hz. We were quite pleased with this result, as this allowed us to devote our time to developing new displays rather than concerning ourselves with producing efficient code for real time animation.

Initial graphics software development was performed on an IRIS 2400 Turbo system that was on-loan from another group. There were few problems encountered in porting software from the older 2400 to the new 4D series machine. There were some delays attributable to differences in the operating systems (4.2 bsd UNIX on the 2400 versus System V UNIX on the 4D). In some cases identical commands require different command options to produce the same results. Some critical system files are stored in different parts of the file system. This is a relatively minor problem but it has continued to be an annoyance as developers utilize both operating systems while working on a variety of machines. Standardization of file systems and command options would be helpful.

The port of graphics software from the 2400 to the 4D machine required few modifications. Some additional code was required to interface with the window manager resident on the 4D that was not present on the 2400. One problem that escaped detection for a long period of time was related to the ease of porting. The 4D added several new capabilities including Gouraud shading of polygons. The graphics system incorporates defaults for all options, and by default, polygon shading was enabled. This incurs a substantial penalty in graphics performance, but it was not visible on the initial port from the 2400 due to the greatly increased speed of the 4D. A detailed study of differences between the machines was not performed as the port appeared to be completed without difficulty. As the software increased in capability and graphics performance demands, we realized that we were not getting the polygon display performance that we had expected. When we discovered the problem and disabled shading, we found that rendering time for large polygons decreased by a factor of over six. It should be noted that the GT enhanced graphics board for the 4D machines performs shading in hardware, and there is no performance penalty for use of this feature.

The software for this project was written in the FORTRAN and C programming languages. The majority of graphics code was written in FORTRAN, while most of the simulation code was written in C. The choice of language was partly based on the perceived suitability of each language to the task at hand, and partly due to the personal preferences and familiarity of the programmers involved. Comparisons of the compute throughput of the two languages were not performed. There were no significant problems encountered in passing data or control of execution between routines written in the two languages. SGI indigenous graphics functions were used in order to extract the maximum possible performance from the workstation.

UNIX development tools were utilized extensively and enhanced both the development rate and quality of the software produced. This would be taken for granted in many programming shops, but such tools are frequently not used in flight simulation environments. Simulation programmers will often use optimization techniques and "tricks" that preclude the use of tools such as symbolic debuggers and performance profilers. Software can be debugged and refined much more quickly with these tools than without them, and the end result is the production of more efficient, readable, and maintainable code.

Simulation

The part-task simulation is set up to operate autonomously. It flies a specified route and approach to an airport in a realistic fashion without pilot interaction. Closed loop control, i.e., manual flying of the aircraft by the flight crew, was not a desired capability, and in fact would distract experimental subjects from the task of interacting with and evaluating the weather displays. This simulates real world operations procedures wherein one pilot flies the aircraft while the other assists by collecting, disseminating and analyzing data relating to the overall flight situation. The computer serves as the pilot flying the aircraft while the experimental subject acts as the pilot not-flying. The subject is thus able to devote full attention to evaluation of weather displays and other flight parameters.

The aircraft simulation is organized as a bare aircraft flight model partially hidden beneath a multi-layered controller model. Yet another layer, the event scheduler, sits on top of the controller.

The aircraft model is based on an older model of a Douglas DC-8. It uses the standard method of linearized partial differential equations for the aircraft kinematics and a large set of coefficients derived from wind tunnel studies which describe the aerodynamic response of the aircraft to its kinematic state. On top of this are superimposed several non-linear effects. The model was extensively re-worked to make it modular and extensible so that, among other things, the type of aircraft can be readily changed by substituting new matrices of coefficients.

The first layer consists of what might be called the stability controller. This is a relatively simple simulation of the stability-augmentation devices found on actual aircraft, such as the yaw damper, which are required to counteract certain instabilities inherent in the swept-wing design of jet airliners.

The next two layers, together called the flight controller, perform functions at the level of the autopilot. The first or lower-level layer uses feedback from the velocities and accelerations of the aircraft model to try and maintain straight and level flight at a given airspeed. Commands from higher levels can specify different control objectives. The second of the two layers tries to maintain an assigned altitude and heading, and will cause the aircraft model to climb or descend at a steady angle and turn at a steady rate until close to the target values. It will then smoothly converge to those values.

The top controller layer, here called the course controller, is presented with a linked list of waypoints, represented as x,y, coordinates (take from latitude/longitude pairs) along with the altitude and wind vector, and flies the aircraft smoothly along the course connecting the waypoints. The logic was tuned to reject or otherwise gracefully handle waypoint sequences which were obviously unreasonable. In principal, the waypoints (beyond the "current" one) can be changed during a run, but this capability was not tested. Additional logic was included at this level to handle the special case of final approach to the runway.

A general-purpose event scheduler was developed so that an arbitrary sequence of events such as aircraft behavior, display of ATC commands or weather, can be pre-programmed to occur at specific times throughout the simulation. These events can be specified as a list of entries in a text file. These entries are converted into a form in which they can be inserted into a sorted queue. When the time arrives for the event at the head of the queue, the element is removed from the queue and an action routine is invoked. The action routine is related to the type of event. It was found that a small number of types sufficed for all needed events, so new action routines were not often required.

Attention was given to keeping the simulation and graphics elements as separate and independent as possible. A small well-defined interface was developed which gathered, scaled and converted the parameters which needed to be shared between these two parts. This was motivated by the concern that for performance reasons, it might be necessary to run the simulation and graphics on two separate processors, or to use shared memory for the interface. As it turned out, due to the power of the single processor system, neither of these has proved necessary to date. The advantages of modularity remain and provide the capability to easily migrate to a multiple processor workstation (such as the IRIS Power Series) in the event that compute requirements grow to exceed that of a single processor.

This simulation code was developed on a DEC VAX 750 mainframe. It executed on the 4D-70 version of the workstation at approximately an order of magnitude greater speed than on the VAX.

Flight Displays

The primary flight instrument display format was patterned after an experimental layout for the Boeing 747-400 and formats currently in use in the ACFS at MVSRF. The basic flight instrument group included precision displays for attitude, airspeed, altitude, instantaneous vertical speed, and course and glideslope deviation indicators.

A combined navigation and airborne weather radar display comprises the remainder of the primary flight instruments. This display featured a moving-map navigation presentation overlayed on simulated airborne weather radar returns. This presentation shows aircraft position and orientation relative to a desired course line and selected navigation fixes. A symbol representing the aircraft is fixed at the bottom center of the display. The map changes orientation as the aircraft moves and turns. This is known as a "heading up" display, as the top of the screen always represents the nose position or heading of the aircraft.

Several parameters are presented digitally, including true airspeed, ground speed, wind speed, and distance, course and time to the next navigation fix. There are sufficient cues provided in these displays to operate and navigate an aircraft with no out-the-window visual references. This is an essential feature of all modern-day air transports, as they must be capable of operation while in or above clouds where there are no visual references to the outside world.

The secondary displays are made up of a text data link display and a second weather display. The textual display is primarily used for up-link of weather and ATC information to the aircraft as an adjunct to voice communications. The weather display was the focus of this research. It was used to test various display concepts for the up-link of weather data obtained by ground based radar systems.

One of the formats tested presents a fixed plan view of the airport and surrounding area with north always at the top of the display. This is referred to as a "north up" display. Returns from ground based radars are superimposed on the plan view. The position and heading of the aircraft relative to the airport are also displayed. This format allows crews to survey the weather patterns in the vicinity of the airport without regard to their position or heading and is intended to facilitate tactical planning for severe weather avoidance. The research was concerned with the effectiveness of this kind of display in alerting pilots to severe weather phenomena that are a threat to the flight safety of aircraft.

Display configuration is table driven. Positions and sizes of display elements are read from ASCII data files at run time. Displays can be rapidly reconfigured during run time to any pre-defined display layout.

Interactive control of navigation and weather displays was patterned after systems in use on current advanced technology aircraft. The controls consisted of a number of dedicated function buttons to choose among range scales and to view or suppress display of weather radar returns. Hazardous weather warning symbols could not be suppressed by the subjects. Each button press is recorded with the simulation time, altitude, position relative to the next navagational fix, and position relative to the airport. These data are analyzed after the completion of the simulation runs to help draw conclusions about how such displays would be used on board actual aircraft, and how effective they might be in preventing encounters with dangerous weather phenomenon.

Graphics Techniques

Graphics programming of these displays was relatively straightforward, but the software does take advantage of several specific features of the IRIS design. A few techniques deserve specific mention.

All displays are double buffered; two sets of graphics image planes (display buffers) are used alternately. One graphics buffer is displayed while the other is being updated, thus the drawing process is hidden from the user and display changes appear to occur smoothly.

The number of colors that can be used effectively for color coding is fairly small. Typically eight or fewer colors are used to code information on instrument displays. Thus full color capabilities are not required, and the available bit planes can be used for other purposes. A color table is used to map the contents of the image memory to colors displayed on the screen. Careful manipulation of the color table and controlled writing to specific bit planes allows creation of display masks and additional foreground / background buffering of the display.

Performance gains can be realized by taking advantage of the frame to frame coherency of the displays. The use of overlay masks allows the creation of transparent "keyholes" or clipping windows. A foreground mask can remain fixed in relation to the screen while background information is rotated and translated. Complex intersections of shapes can thus be displayed without having to compute their intersections. This technique is used in the construction of the attitude indicator and airborne weather displays.

In some cases the bit planes in each buffer are treated as two to four sets. Foreground and background images can be created and revised without effecting each other, and images can effectively be double buffered within each buffer. This technique can be used to enable differing rates of updates of the screen elements. Display components can be updated at a rate different from the frame buffer swapping rate and are hidden from view until their drawing is completed. Foreground / background switching is performed by manipulating the color table to hide or show the contents of the desired planes.

The above techniques were combined in the ACFS to allow airborne radar weather updates to be performed at a slower rate than for the aircraft control and navigation instruments. New radar images are built and refreshed at approximately four Hz while the remainder of the same display is refreshed at 30 Hz.

Integration with Full Scale Simulators

It was originally intended that selected displays from the part task simulations would be re-coded for operation on the existing hardware in the ACFS. However, it was determined that the graphics and compute demands of the new format displays would strain the capabilities of this equipment. Consequently it was decided to integrate an IRIS 4D-80 GT workstation into the ACFS in parallel to the existing hardware. Integration was facilitated by the capability of the IRIS to drive a variety of monitor types, including the type that was originally installed in the flight simulator.

The original "quick and dirty" primary flight display code was pressed into service for this application and enhanced by MVSRF staff well before the part-task simulation itself was completed. The display capabilities of the 4D-80 GT system enabled the new displays to significantly out-perform the original equipment, even though the original display code was optimized. This reflects the large increase in performance of the IRIS over previous generation graphics hardware.

Integration of the IRIS and modification of the part-task software required considerably less time than would have been required for development of similar displays on the original simulator equipment. The common hardware base now provides a direct path from the experimental lab setting to the full flight simulation environment.

Pilots who flew the simulator reported no problems in manually flying the aircraft by reference to these flight instrument displays. They also stated that the workstation graphics operated nearly identically to actual flight hardware, thus it was very easy for them to adapt to the simulator environment.

Simulating a Flight

The simulation of a flight requires the combination of all the software discussed above with a series of data files that define the flight scenario. All of the data files are stored in ASCII to permit easy examination and modification of the flight profile.

The first step is to select a geographical area for the simulated flight. Navigation fixes must be selected for inclusion on navigation and weather displays. A plan view of the vicinity of the airport is created by producing a list of graphic primitives to show critical navigation references and runway alignment. Displays are created at several resolutions as the required information changes with distance from the airport.

The next step is to define a route of flight and how it is to be flown. A linked list of navigational fixes describes the route. Each point is specified in degrees of latitude and longitude and assigned a target airspeed and altitude which the pilot model seeks. A typical route involves a short cruise segment followed by a descent for an approach to an airport. Routes are typically designed to include an approach to landing followed by a go-around maneuver (climb to altitude for another approach).

Weather patterns are defined through the use of a raster-oriented paint program and vector conversion software to generate images for the ground and airborne weather radar displays, respectively. The formats reflect the types of displays that are likely to be installed on actual aircraft. Weather patterns can be time-sequenced to simulate a storm pattern that changes over time.

Finally, an event time-line is created that sequences the display of weather patterns, alerts, and ATC messages. Display contents can be changed via the time sequence or through manual selection by the subject.

When the simulation is invoked, the sequencer causes each data file to be read at the specified time. The aircraft and pilot models calculate aircraft position and orientation, and the displays are refreshed at the end of each compute cycle to reflect the current conditions. The subject observes the progress of the flight and selects weather products as desired in order to assess the flight situation. The pilot uses the information to identify critical weather phenomena and thus decide whether to continue the flight through landing or at which point to abort the approach.

Together, these elements combine to create a realistic decision making environment for the crew-member subjects. Their interactions with the secondary displays is recorded and analyzed in order to develop an understanding of how such displays would be used and how they could be made more effective.

Future Directions

The software developed for this project will serve as a base for additional part-task studies of flight display systems. The commonality of hardware between the lab and full scale flight simulator environments will increase the speed and decrease the cost and technical difficulty in transporting new displays from conceptual stages to testing in the full mission environment.

There is no requirement for the display software to be driven exclusively by flight simulation software. They could be driven by performance data taken from actual aircraft in flight thus allowing detailed analysis of flight displays on the ground. This could be of value in aircraft accident analysis. Safety investigators could use data from an aircraft's flight data recorder (the "black box") to drive the displays and see the flight instruments as they would have appeared to the flight crew of an accident aircraft.

The decreasing cost of of computer graphics workstations of this class will allow a larger number of groups to utilize these capabilities. Hopefully this will lead to increased understanding of modern technology aircraft systems and have an attendant benefit to flight safety.

Acknowledgements

The authors thank the following NASA Ames Research Center personnel who have major responsibility for research work discussed in this paper. Alfred T. Lee, PhD, Research Psychologist, Aviation Systems Research Branch, R. Curtis Graeber, PhD, Chief, Aviation Systems Research Branch, and Robert J. Shiner, Manager, Man-Vehicle Systems Research Facility.

The Shape of PSIBER Space: PostScript Interactive Bug Eradication Routines

Don Hopkins

don@brillig.umd.edu
University of Maryland
Human-Computer Interaction Lab
Computer Science Department
College Park, Maryland 20742

ABSTRACT

The PSIBER Space Deck is an interactive visual user interface to a graphical programming environment, the NeWS window system. It lets you display, manipulate, and navigate the data structures, programs, and processes living in the virtual memory space of NeWS. It is useful as a debugging tool, and as a hands on way to learn about programming in PostScript and NeWS.

1. INTRODUCTION

''Cyberspace. A consensual hallucination experienced daily by billions of legitimate operators, in every nation, by children being taught mathematical concepts ... A graphic representation of data abstracted from the banks of every computer in the human system. Unthinkable complexity. Lines of light ranged in the nonspace of the mind, clusters and constellations of data. Like city lights, receding ....''

[Gibson, Neuromancer]

The PSIBER Space Deck is a programming tool that lets you graphically display, manipulate, and navigate the many PostScript data structures, programs, and processes living in the virtual memory space of NeWS.

The Network extensible Window System (NeWS) is a multitasking object oriented PostScript programming environment. NeWS programs and data structures make up the window system kernel, the user interface toolkit, and even entire applications.

The PSIBER Space Deck is one such application, written entirely in PostScript, the result of an experiment in using a graphical programming environment to construct an interactive visual user interface to itself.

It displays views of structured data objects in overlapping windows that can be moved around on the screen, and manipulated with the mouse: you can copy and paste data structures from place to place, execute them, edit them, open up compound objects to see their internal structure, adjust the scale to shrink or magnify parts of the display, and pop up menus of other useful commands. Deep or complex data structures can be more easily grasped by applying various views to them.

There is a text window onto a NeWS process, a PostScript interpreter with which you can interact (as with an ''executive''). PostScript is a stack based language, so the window has a spike sticking up out of it, representing the process's operand stack. Objects on the process's stack are displayed in windows with their tabs pinned on the spike. (See figure 1) You can feed PostScript expressions to the interpreter by typing them with the keyboard, or pointing and clicking at them with the mouse, and the stack display will be dynamically updated to show the results.

Not only can you examine and manipulate the objects on the stack, but you can also manipulate the stack directly with the mouse. You can drag the objects up and down the spike to change their order on the stack, and drag them on and off the spike to push and pop them; you can take objects off the spike and set them aside to refer to later, or close them into icons so they don't take up as much screen space.

NeWS processes running in the same window server can be debugged using the existing NeWS debug commands in harmony with the graphical stack and object display.

The PSIBER Space Deck can be used as a hands on way to learn about programming in PostScript and NeWS. You can try out examples from cookbooks and manuals, and explore and enrich your understanding of the environment with the help of the interactive data structure display.

2. INTERACTING WITH DATA

A PostScript object is a reference to any piece of data, that you can push onto the stack. (The word "object" is used here in a more general sense than in "object oriented programming." The words "class" and "instance" are used for those concepts.) Each object has a type, some attributes, and a value. PostScript objects are dynamically typed, like Lisp objects, not statically typed, like C variables. Each object is either literal or executable. This attribute effects whether the interpreter treats it as data or instructions. Compound objects, such as arrays and dictionaries, can contain references to other objects of any type. [Adobe, Red, Green, and Blue books] [Sun, NeWS 1.1 Manual] [Gosling, The NeWS Book]

2.1. Viewing Data

Objects on the PSIBER Space Deck appear in overlapping windows, with labeled tabs sticking out of them. Each object has a label, denoting its type and value, i.e. "integer 42". Each window tab shows the type of the object directly contained in the window. Objects nested within other objects have their type displayed to the left of their value. The labels of executable objects are displayed in italics.

2.1.1. Simple Objects

Figure 1 shows some simple objects: an integer, a boolean, a literal string, an executable string, a literal name, and an executable name -- the results of executing the string "42 false (flamingo) (45 cos 60 mul) cvx /foobar /executive cvx".

Strings are delimited by parenthesis: "string (flamingo)". Literal names are displayed with a slash before them: "name /foobar". Executable names are displayed in italics, without a leading slash: *"name executive"*. Names are most often used as keys in dictionaries, associated with the values of variables and procedures.

2.1.2. Compound Objects

Compound objects, which contain other objects, can be displayed closed or opened. The two basic kinds of compound objects are arrays and dictionaries. Arrays are indexed by integers, and dictionaries are indexed by keys.

Figure 2 shows a literal array, an executable array, and a dictionary.

An opened compound object is drawn with lines fanning out to the right, leading from the object to its elements, which are labeled as ''index: type value'', in a smaller point size.

Literal arrays are displayed with their length enclosed in square brackets: ''array [6]''. Executable arrays (procedures) are displayed in italics, with their length enclosed in braces: *''array {37}''*. The lines fanning out from an opened array to its elements are graphically embraced, so they resemble the square brackets or braces in the label of a literal or executable array.

PostScript arrays are polymorphic: Each array element can be an object of any type. A PostScript procedure is just an executable array of other objects, to be interpreted one after the other.

The label of a dictionary shows the number of keys it contains, a slash, and the maximum size of the dictionary, enclosed in angled brackets: ''dict <5/10>''.

The lines that fan out from opened dictionaries resemble the angled brackets appearing in their labels.

Dictionaries associate keys with values. The key (an index into a dictionary) can be an object of any type (except null), but is usually a name. The value can be anything at all. Dictionaries are used to hold the values of variables and function definitions, and as local frames, structures, lookup tables, classes, instances, and lots of other things -- they're very useful!

The dictionary stack defines the scope of a PostScript process. Whenever a name is executed or loaded, the dictionaries on the dictionary stack are searched, in top to bottom order.

2.1.3. Classes, Instances, and Magic Dictionaries

NeWS uses an object oriented PostScript programming package, which represents classes and instances with dictionaries. [Densmore, Object Oriented Programming in NeWS] [Gosling, The NeWS Book]

When a class dictionary is displayed, the class name is shown, instead of the ''dict'' type: ''Object <10/200>''. When an instance dictionary is displayed, its type is shown as a period followed by its class name: ''.SoftMenu <31/200>''.

Figure 3 shows the class dictionary of Object, and the instance dictionary of the NeWS root menu.

Magic dictionaries are certain types of NeWS objects, such as processes, canvases, and events, that act as if they were dictionaries, but have some special internal representation. They have a fixed set of keys with special meanings (such as a process's ''/OperandStack'', or a canvas's ''/TopChild''), that can be accessed with normal dictionary operations. Special things may happen when you read or write the values of the keys (for example, setting the ''/Mapped'' key of a canvas to false makes it immediately disappear from the screen). [Sun, NeWS 1.1 Manual] [Gosling, The NeWS Book]

Figure 4 shows a canvas magic dictionary (the framebuffer), and a process magic dictionary, with some interesting keys opened.

2.1.4. View Characteristics

The views of the objects can be adjusted in various ways. The point size can be changed, and well as the shrink factor by which the point size is multiplied as the structure gets deeper. The point size is not allowed to shrink smaller than 1, so that labels will never have zero area, and it will always be possible to select them with the mouse. If the shrink factor is greater than 1.0, the point size increases with depth.

The nested elements of a compound object can be drawn either to the right of the object label, or indented below it. When the elements are drawn indented below the label, it is not as visually obvious which elements are nested inside of which object, but it takes up a lot less screen space than drawing the elements to the right of the label does.

Any of the view characteristics can be set for a whole window, or for any nested object and its children within that window.

Figure 5 shows some nested structures with various point sizes and shrink factors, with elements opened to the right and below.

2.2. Editing Data

There are many ways to edit the objects displayed on the screen. There are functions on menus for converting from type to type, and for changing the object's attributes and value. You can replace an element of a compound object by selecting another object, and pasting it into the element's slot. There are also a bunch of array manipulation functions, for appending together arrays, breaking them apart, rearranging them, and using them as stacks. You must be careful which objects you edit, because if you accidentally scramble a crucial system function, the whole window system could crash.

2.3. Peripheral Controls

Peripheral controls are associated views that you can attach to an object, which are not directly contained within that object. They are visually distinct from the elements of a compound object, and can be used to attach editor buttons, computed views, and related objects. Several useful peripheral views have been implemented for manipulating various data types.

There are three types of numeric editors: the step editor, the shift editor, and the digit editor. The step editor has ''++'' and ''--'' buttons to increment and decrement the number it's attached to, by the parameter ''Step''. The shift editor has ''**'' and ''//'' buttons to multiply and divide the number it's attached to, by the parameter ''Shift''. The ''Step'' and ''Shift'' parameters appear in the peripheral views as normal editable numbers, to which you can attach other numeric editors, nested as deep as you like. The digit editor behaves like a numeric keypad, with buttons for the digits 0 through 9, ''Rubout'', ''Clear'', and ''+-''.

The boolean editor has ''True'', ''False'', and ''Not'' buttons that do the obvious things, and a ''Random'' button, that sets the boolean value randomly. Since the button functions are just normal data, you can open up the ''Random'' button and edit the probability embedded in the function ''{random 0.5 lt}''.

You can open up a definition editor on a name, to get editable references to every definition of the name on the dictionary stack (or in the context to which the enclosing class editor is attached).

The scroller editor allows you to view a reasonably sized part of a large array or dictionary. The peripheral controls include a status line telling the size of the object and how much of it is shown in the view, ''Back'' and ''Next'' buttons for scrolling the view, and a ''Size'' parameter that controls the number of elements in the view. You can edit the ''Size'' parameter of the scrolling view by attaching a numeric editor to it, or dropping another number into its slot, and it will take effect next time you scroll the view.

When you open a class editor, it attaches the following peripheral views: ''ClassDicts'', an array of the class dictionaries, ''SubClasses'', an array of a class's subclasses, ''InstanceVars'', an array of instance variable names, ''ClassVars'', an array class variable names, and ''Methods'', an array of method names. You can open up scrolling views on the arrays, and open up definition editors on the names, and you will be able to examine and edit the definitions in the class.

The canvas editor gives you a graphical view of the canvas's relation to its parent, and an array of the canvas's children. You can grab the graphical view of the canvas with the mouse and move the canvas itself around. You can open up the array of child canvases (with a scroller editor if you like), and attach canvas views to them, too.

Figure 6 shows some digit editors, step editors, shift editors, a boolean editor, and a canvas editor. Figure 7 shows a class editor, some scroller editors, name editors, and digit editors.

2.4. Printing Distilled PostScript

The data structure displays (including those of the Pseudo Scientific Visualizer, described below) can be printed on a PostScript printer by capturing the drawing commands in a file.

Glenn Reid's ''Distillery'' program is a PostScript optimizer, that executes a page description, and (in most cases) produces another smaller, more efficient PostScript program, that prints the same image. [Reid, The Distillery] The trick is to redefine the path consuming operators, like fill, stroke, and show, so they write out the path in device space, and incremental changes to the graphics state. Even though the program that computes the display may be quite complicated, the distilled graphical output is very simple and low level, with all the loops unrolled.

The NeWS distillery uses the same basic technique as Glenn Reid's Distillery, but it is much simpler, does not optimize as much, and is not as complete.

3. INTERACTING WITH THE INTERPRETER

In PostScript, as in Lisp, instructions and data are made out of the same stuff. One of the many interesting implications is that tools for manipulating data structures can be used on programs as well.

3.1. Using the Keyboard

You can type PostScript expressions into a scrolling text window, and interact with the traditional PostScript ''executive,'' as you can with ''psh'' to NeWS or ''tip'' to a laser printer. Certain function keys and control characters do things immediately when you type them, such as input editing, selecting the input, pushing or executing the selection, and completing names over the dictionary stack (like ''tcsh'' file name completion).

3.2. Using the Mouse

The mouse can be used to select data, push it on the stack, execute it, and manipulate it in many ways.

Pointing the cursor at an object and clicking the ''Menu'' button pops up a menu of operations that can be performed on it. All data types have the same top level pop-up menu (for uniformity), with a type specific submenu (for diversity). There are lots of commands for manipulating the object and the view available via pop-up menus.

You can select any object by clicking the ''Point'' button on it. A printed representation of the current selection is always displayed in a field at the top of the scrolling text window. If you click the Point button over an object whose label is too small to read, it will appear in the selection field, in a comfortable font.

Each object has its own button handler function that is called when you click the ''Adjust'' button on it. The default ''Adjust'' handler implements ''drag'n'dropping''. If you drop an object onto itself, its view toggles open or closed. If you drop it on top of a compound object element, it is stored into that memory location. If you drop it over an unoccupied spot, a new window viewing the object appears on the deck.

Another useful ''Adjust'' handler simply executes the object that was clicked on. This can be used to make buttons out of executable names, arrays, and strings.

3.3. Using Dictionaries as Command Pallets

A PostScript dictionary can be used as a pallet of commands, by defining a bunch of useful functions in a dictionary, opening it up, and executing the functions with the mouse. You can open up the functions to see their instructions, and even edit them!

3.4. Using a Text Editor

It is very helpful to be running a text editor on the source code of a PostScript program, while you are debugging it. You can select chunks of source from the text editor, and execute them in the PSIBER Space Deck (in the appropriate context). This is especially useful for redefining functions of a running program in memory, as bugs are discovered and fixed in the source code. It saves you from having to kill and restart your application every time you find a trivial bug.

4. DEBUGGING PROGRAMS

The NeWS debugger lets you take on and examine the state of a broken process. [Sun, NeWS 1.1 Manual] [Gosling, The NeWS Book] The debugger is a PostScript program originally written for use with ''psh'' from a terminal emulator. It is notorious for being difficult to use, but quite powerful. However, the debugger is much nicer in conjunction with the graphical stack, the object display, and a pallet of handy debugging commands, that you can invoke with the mouse.

When you enter a broken process with the debugger, you get a copy of its operand and dictionary stacks. You can examine and manipulate the objects on the stack, look at the definitions of functions and variables, and execute instructions in the scope of the broken process. You can change the stack, copy it back, and continue the process, or just kill it. You can push onto the spike the broken process, its dictionary stack, and its execution stack, and open them up to examine the process's state.

I use the debugger extensively in developing the PSIBER Space Deck, both from a terminal emulator and from the deck itself. Using the deck to debug itself is an interesting experience. One of the most common uses is to redefine a function by selecting some text in from Emacs and executing it. But it's still easy to make a mistake and crash it.

5. THE USER INTERFACE

5.1. Pie Menus

The mouse button functions and menu layouts were designed to facilitate gestural interaction, to simulate the feel of tweaking and poking at real live data structures.

There are several ''pull out'' pie menus, that use the cursor distance from the menu center as an argument to the selection.

The pie menu that pops up over data objects has the commonly used functions ''Push,'' ''Exec,'' and ''Paste'' positioned in easily selected directions (up, down, and left). Once you are familiar enough with the interface to ''mouse ahead'' into the menus, with quick strokes of the mouse in the appropriate direction, interaction can be very swift. [Callahan, A Comparative Analysis of Pie Menu Performance] [Hopkins, A Pie Menu Cookbook]

When you mouse ahead through a pie menu selection quickly enough, the menu is not displayed, and the shape of a pac-man briefly flashes on the screen, with its mouth pointing in the direction of the selected menu item. This ''mouse ahead display suppression'' speeds up interaction considerably by avoiding unnecessary menu display, and makes it practically impossible for the casual observer to follow what is going on. The flashing pac-man effect gives you some computationally inexpensive feedback of the menu selection, and reassures observers that you are racking up lots of points.

5.2. Tab Windows

The objects on the deck are displayed in windows with labeled tabs sticking out of them, showing the data type of the object. You can move an object around by grabbing its tab with the mouse and dragging it. You can perform direct stack manipulation, pushing it onto stack by dragging its tab onto the spike, and changing its place on the stack by dragging it up and down the spike. It implements a mutant form of ''Snap-dragging'', that constrains non-vertical movement when an object is snapped onto the stack, but allows you to pop it off by pulling it far enough away or lifting it off the top. [Bier, Snap-dragging] The menu that pops up over the tab lets you do things to the whole window, like changing view characteristics, moving the tab around, repainting or recomputing the layout, and printing the view.

6. THE METACIRCULAR POSTSCRIPT INTERPRETER

A program that interprets the language it is written in is said to be ''metacircular''. [Abelson, Structure and Interpretation of Computer Programs] Since PostScript, like Scheme, is a simple yet powerful language, with procedures as first class data structures, implementing ''ps.ps'', a metacircular PostScript interpreter, turned out to be straightforward (or drawrofthgiarts, with respect to the syntax). A metacircular PostScript interpreter should be compatible with the ''exec'' operator (modulo bugs and limitations). Some of the key ideas came from Crispin Goswell's PostScript implementation. [Goswell, An Implementation of PostScript]

The metacircular interpreter can be used as a debugging tool, to trace and single step through the execution of PostScript instructions. It calls a trace function before each instruction, that you can redefine to trace the execution in any way. One useful trace function animates the graphical stack on the PSIBER Space Deck step by step.

The meta-execution stack is a PostScript array, into which the metacircular interpreter pushes continuations for control structures. (forall, loop, stopped, etc...) A continuation is represented as a dictionary in which the state needed by the control structure is stored (plus some other information to help with debugging).

It is written in such a way that it can interpret itself: It has its own meta-execution stack to store the program's state, and it stashes its own state on the execution stack of the interpreter that's interpreting it, so the meta-interpreter's state does not get in the way of the program it's interpreting.

It is possible to experiment with modifications and extensions to PostScript, by revectoring functions and operators, and modifying the metacircular interpreter.

The metacircular interpreter can serve as a basis for PostScript algorithm animation. One very simple animation is a two dimensional plot of the operand stack depth (x), against the execution stack depth (y), over time.

7. THE PSEUDO SCIENTIFIC VISUALIZER

"Darkness fell in from every side, a sphere of singing black, pressure on the extended crystal nerves of the universe of data he had nearly become... And when he was nothing, compressed at the heart of all that dark, there came a point where the dark could be no *more*, and something tore. The Kuang program spurted from tarnished cloud, Case's consciousness divided like beads of mercury, arcing above an endless beach the color of the dark silver clouds. His vision was spherical, as though a single retina lined the inner surface of a globe that contained all things, if all things could be counted. "

[Gibson, Neuromancer]

The Pseudo Scientific Visualizer is the object browser for the other half of your brain, a fish-eye lens for the macroscopic examination of data. It can display arbitrarily large, arbitrarily deep structures, in a fixed amount of space. It shows form, texture, density, depth, fan out, and complexity.

It draws a compound object as a circle, then recursively draws its elements, scaled smaller, in an evenly spaced ring, rotated around the circle. The deeper an object, the smaller it is. It will only draw to a certain depth, which you can change while the drawing is in progress.

It has simple graphical icons for different data types. An array is a circle, and a dictionary is a circle with a dot. The icon for a string is a line, whose length depends on the length of the string. A name is a triangle. A boolean is a peace sign or an international no sign. An event is an envelope. A process is a Porsche.

It randomly forks off several light weight processes, to draw different parts of the display, so there is lots of drawing going on in different places at once, and the overlapping is less regular.

After the drawing is complete, the circular compound objects become mouse sensitive, selectable targets. The targets are implemented as round transparent NeWS canvases. When you move the cursor over one, it highlights, and you can click on it to zoom in, pop up a description of it, open up another view of it, or select it, and then push it onto the stack of the PSIBER Space Deck.

Figure 8 shows a Pseudo Scientific visualization of the NeWS rootmenu instance dictionary, also shown in figure 3 and figure 7. Figure 9 shows two views of a map of the ARPAnet. Figure 10 shows two views of a map of Adventure.

8. REFERENCES

Abelson, Harold; Sussman, Gerald

Structure and Interpretation of Computer Programs; 1985; The MIT Press, Cambridge, Mass. and McGraw Hill, New York

Adobe Systems

PostScript Language Tutorial and Cookbook (The Blue Book); 1985; Addison-Wesley Publishing Company, Inc., Reading, Mass.

PostScript Language Reference Manual (The Red Book); 1985; Addison-Wesley Publishing Company, Inc., Reading, Mass.

Adobe Systems; Reid, Glenn C.

PostScript Language Program Design (The Green Book); 1988; Addison-Wesley Publishing Company, Inc., Reading, Mass.

Bier, Eric A.; Stone, Maureen

Snap-dragging; SIGGRAPH'86 Proceedings; Page 233-240; 1986; ACM, New York

Callahan, Jack; Hopkins, Don; Weiser, Mark; Shneiderman, Ben

A Comparative Analysis of Pie Menu Performance; Proc. CHI'88 conference, Washington D.C.; 1988; ACM, New York

Densmore, Owen

Object Oriented Programming in NeWS; November 1986; USENIX Monterey Computer Graphics Workshop; Usenix Association

Gibson, William

Neuromancer; 1984; ACE Science Fiction Books; The Berkley Publishing Group, New York

Gosling, James; Rosenthal, David S.H.; Arden, Michelle

The NeWS Book; 1989; Springer-Verlag, New York

Goswell, Crispin

''An Implementation of PostScript'', in Workstations and Publication Systems; Rae A. Earnshaw, Editor; 1987; Springer-Verlag, New York

Hopkins, Don

''Directional Selection is Easy as Pie Menus!'', in ;login: The USENIX Association Newsletter; Volume 12, Number 5; September/October 1987; Page 31

A Pie Menu Cookbook: Techniques for the Design of Circular Menus; (Paper in preparation. Draft available from author.)

Hopkins, Don; Callahan, Jack; Weiser, Mark

Pies: Implementation, Evaluation, and Application of Circular Menus; (Paper in preparation. Draft available from authors.)

Reid, Glenn

The Distillery (program); available from ps-file-server@adobe.com

Shu, Nan C.

Visual Programming; 1988; Van Nostrand Reinhold; New York

Sun Microsystems

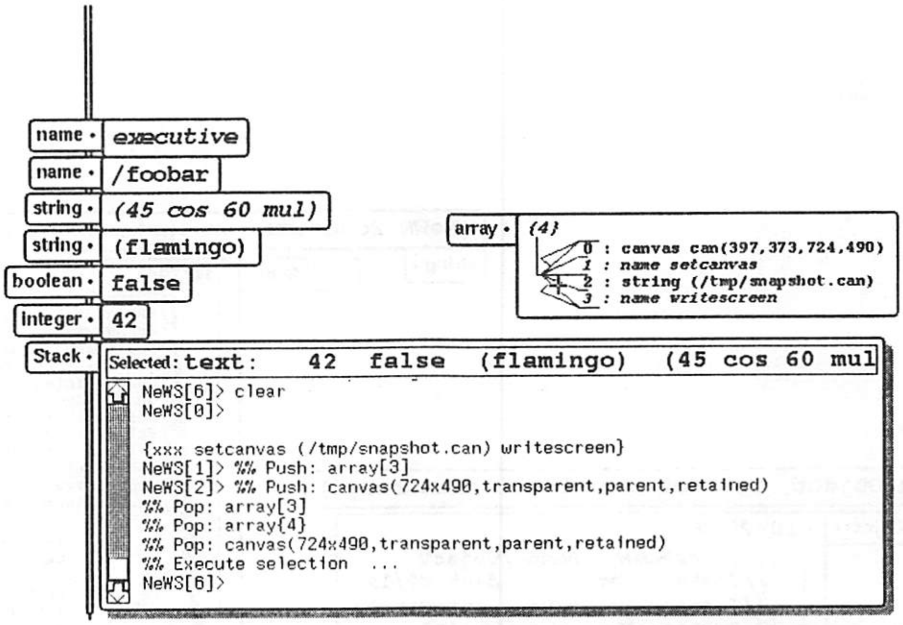
NeWS 1.1 Manual; 1987; Sun Microsystems; Mountain View, California

9. ACKNOWLEDGMENTS

This work could not have been done without the greatly appreciated support of the University of Maryland Human-Computer Interaction Lab, Grasshopper Group, Sun Microsystems, and NCR Corporation.

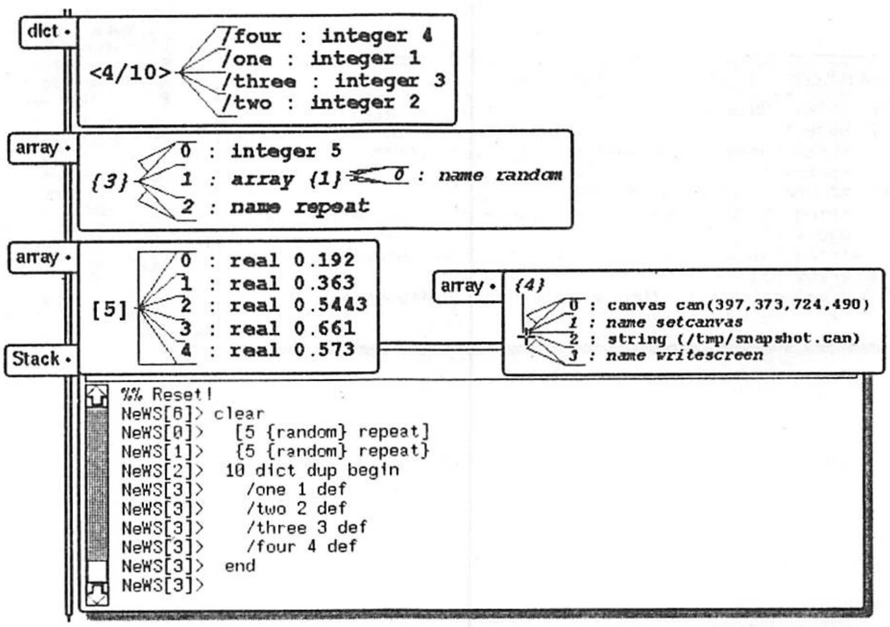
Many thanks to Owen Densmore, Hugh Daniel, Mike Gallaher, John Gilmore, James Gosling, Marja Koivunen, David LaVallee, Julia Menapace, Rehmi Post, Brian Raymor, Glenn Reid, David Rosenthal, Robin Schaufler, Ben Shneiderman, Josh Siegel, Stan Switzer, and Martha Zimet, people who gave me much valuable advice, encouragement, feedback, and lots of neat ideas.

Figure 1: Simple Objects



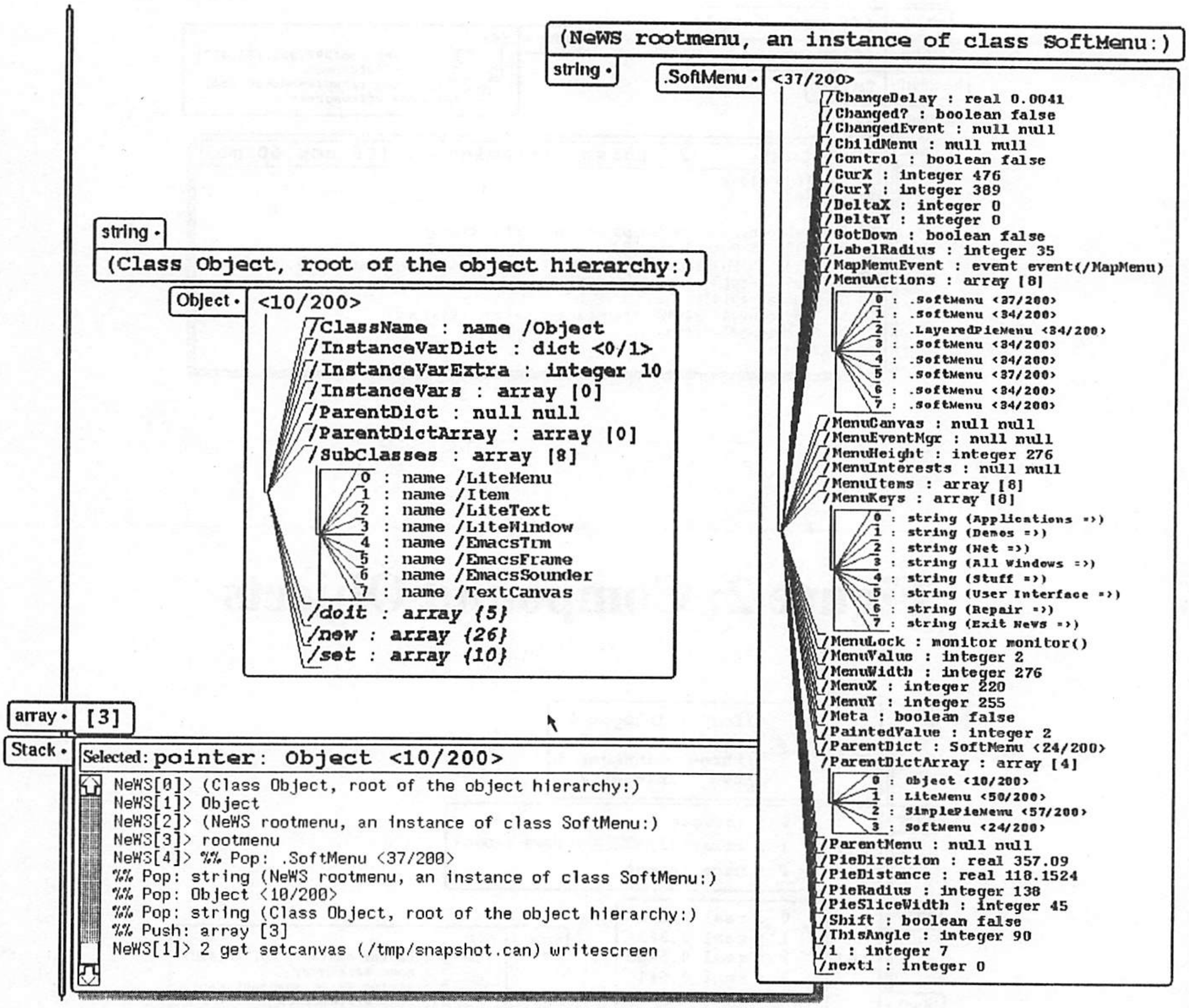
```
executive
/foobar
(45 cos 60 mul)
(flamingo)
false
42

{4}
  0 : canvas can(397,373,724,490)
  1 : name setcanvas
  2 : string (/tmp/snapshot.can)
  3 : name writescreen

Selected: text:    42   false   (flamingo)   (45 cos 60 mul
NeWS[6]> clear
NeWS[0]>

{xxx setcanvas (/tmp/snapshot.can) writescreen}
NeWS[1]> %% Push: array[3]
NeWS[2]> %% Push: canvas(724x490,transparent,parent,retained)
%% Pop: array[3]
%% Pop: array{4}
%% Pop: canvas(724x490,transparent,parent,retained)
%% Execute selection  ...
NeWS[6]>
```

Figure 2: Compound Objects



```
<4/10>
  /four : integer 4
  /one : integer 1
  /three : integer 3
  /two : integer 2

{3}
  0 : integer 5
  1 : array {1} — 0 : name random
  2 : name repeat

[5]
  0 : real 0.192
  1 : real 0.363
  2 : real 0.5443
  3 : real 0.661
  4 : real 0.573

{4}
  0 : canvas can(397,373,724,490)
  1 : name setcanvas
  2 : string (/tmp/snapshot.can)
  3 : name writescreen

%% Reset!
NeWS[6]> clear
NeWS[0]>   [5 {random} repeat]
NeWS[1]>   {5 {random} repeat}
NeWS[2]>  10 dict dup begin
NeWS[3]>     /one 1 def
NeWS[3]>     /two 2 def
NeWS[3]>     /three 3 def
NeWS[3]>     /four 4 def
NeWS[3]>  end
NeWS[3]>
```

Figure 3: Class and Instance

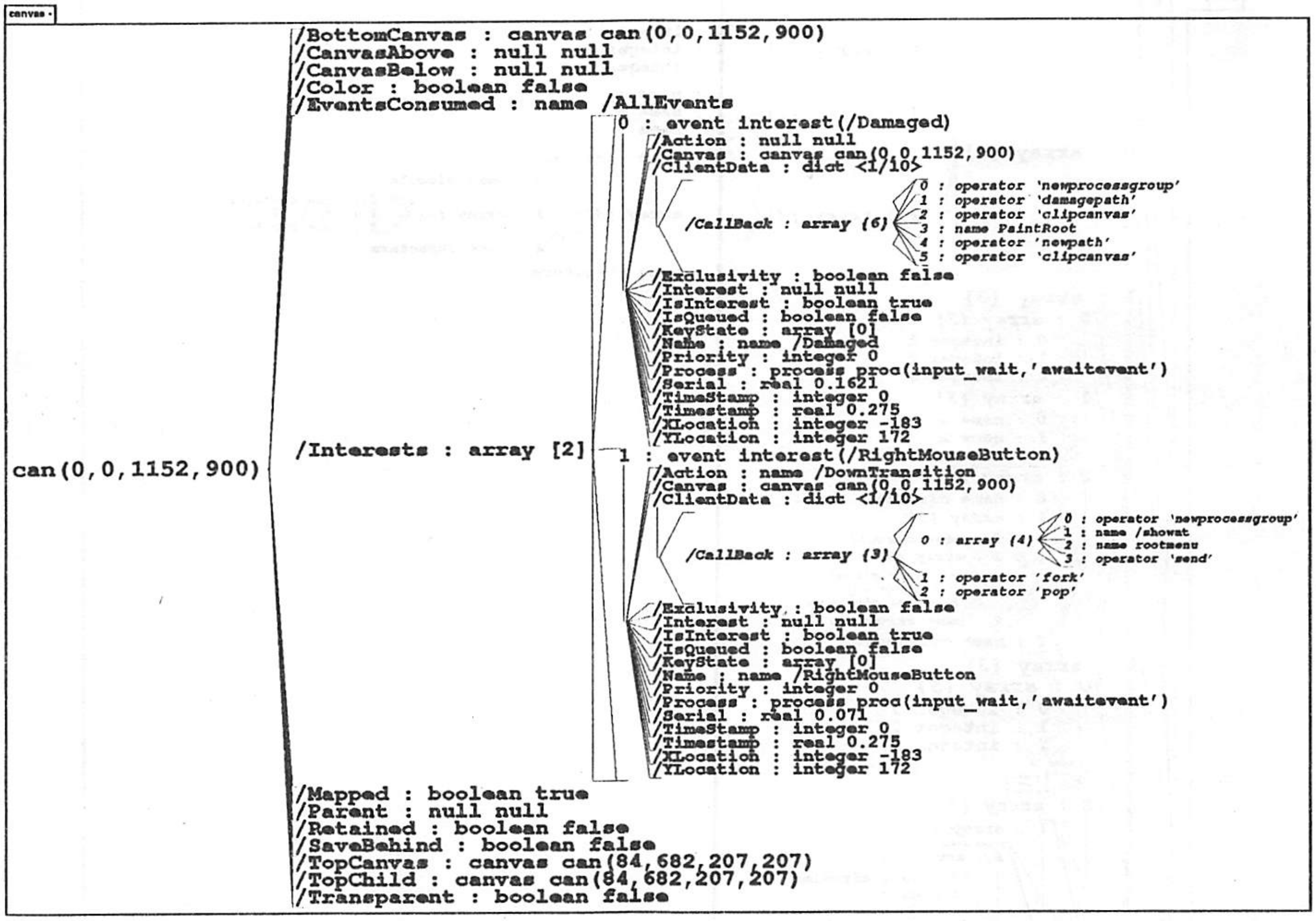


```
(NeWS rootmenu, an instance of class SoftMenu:)

<37/200>
/ChangeDelay : real 0.0041
/Changed? : boolean false
/ChangedEvent : null null
/ChildMenu : null null
/Control : boolean false
/CurX : integer 476
/CurY : integer 389
/DeltaX : integer 0
/DeltaY : integer 0
/GotDown : boolean false
/LabelRadius : integer 35
/MapMenuEvent : event event(/MapMenu)
/MenuActions : array [8]
    0 : .SoftMenu <37/200>
    1 : .SoftMenu <34/200>
    2 : .LayeredPieMenu <34/200>
    3 : .SoftMenu <34/200>
    4 : .SoftMenu <34/200>
    5 : .SoftMenu <37/200>
    6 : .SoftMenu <34/200>
    7 : .SoftMenu <34/200>
/MenuCanvas : null null
/MenuEventMgr : null null
/MenuHeight : integer 276
/MenuInterests : null null
/MenuItems : array [8]
/MenuKeys : array [8]
    0 : string (Applications =>)
    1 : string (Demos =>)
    2 : string (Net =>)
    3 : string (All Windows =>)
    4 : string (Stuff =>)
    5 : string (User Interface =>)
    6 : string (Repair =>)
    7 : string (Exit NeWS =>)
/MenuLock : monitor monitor()
/MenuValue : integer 2
/MenuWidth : integer 276
/MenuX : integer 220
/MenuY : integer 255
/Meta : boolean false
/PaintedValue : integer 2
/ParentDict : SoftMenu <24/200>
/ParentDictArray : array [4]
    0 : Object <10/200>
    1 : LiteMenu <50/200>
    2 : SimplePieMenu <57/200>
    3 : SoftMenu <24/200>
/ParentMenu : null null
/PieDirection : real 357.09
/PieDistance : real 118.1524
/PieRadius : integer 138
/PieSliceWidth : integer 45
/Shift : boolean false
/ThisAngle : integer 90
/i : integer 7
/nexti : integer 0
```

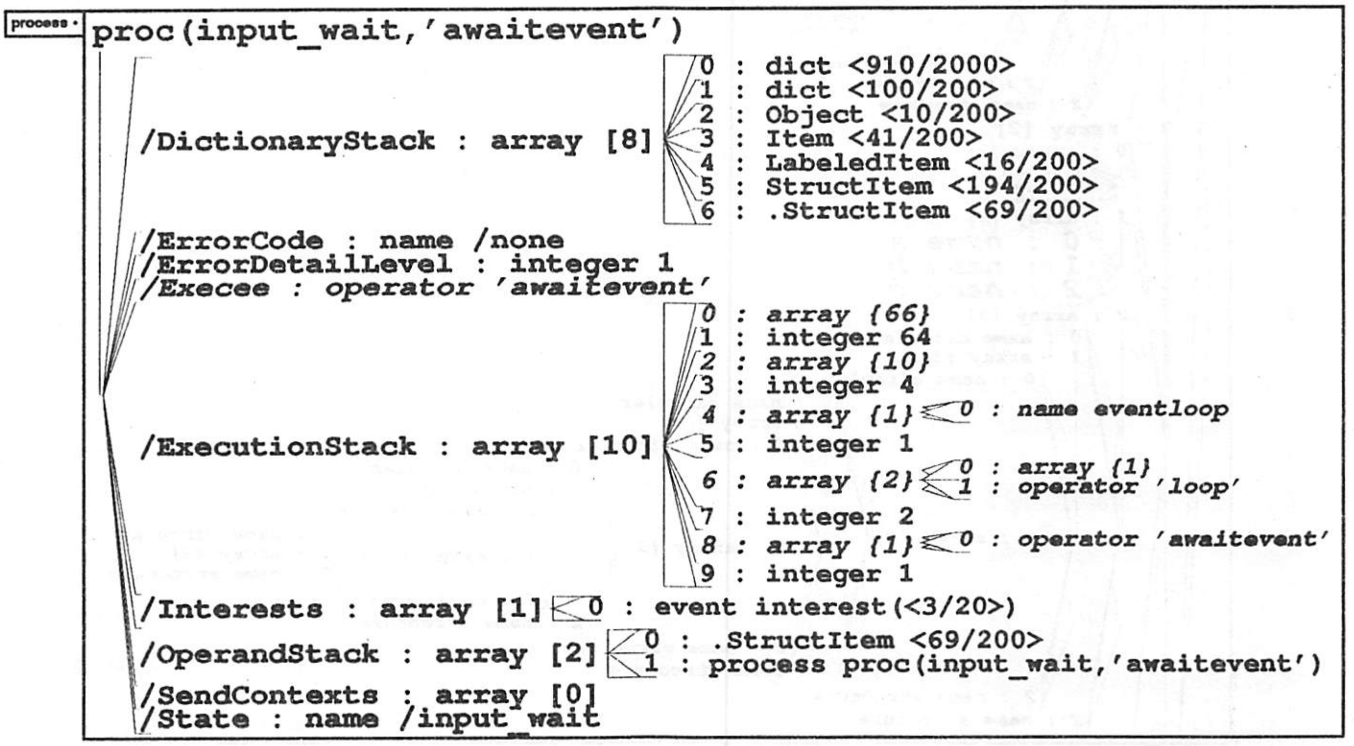
```
(Class Object, root of the object hierarchy:)

<10/200>
/ClassName : name /Object
/InstanceVarDict : dict <0/1>
/InstanceVarExtra : integer 10
/InstanceVars : array [0]
/ParentDict : null null
/ParentDictArray : array [0]
/SubClasses : array [8]
    0 : name /LiteMenu
    1 : name /Item
    2 : name /LiteText
    3 : name /LiteWindow
    4 : name /EmacsTrm
    5 : name /EmacsFrame
    6 : name /EmacsSounder
    7 : name /TextCanvas
/doit : array {5}
/new : array {26}
/set : array {10}
```

```
[3]

Selected: pointer: Object <10/200>
NeWS[0]> (Class Object, root of the object hierarchy:)
NeWS[1]> Object
NeWS[2]> (NeWS rootmenu, an instance of class SoftMenu:)
NeWS[3]> rootmenu
NeWS[4]> %% Pop: .SoftMenu <37/200>
%% Pop: string (NeWS rootmenu, an instance of class SoftMenu:)
%% Pop: Object <10/200>
%% Pop: string (Class Object, root of the object hierarchy:)
%% Push: array [3]
NeWS[1]> 2 get setcanvas (/tmp/snapshot.can) writescreen
```

Figure 4: Magic Dictionaries



```
canvas •
can(0,0,1152,900)
  /BottomCanvas : canvas can(0,0,1152,900)
  /CanvasAbove : null null
  /CanvasBelow : null null
  /Color : boolean false
  /EventsConsumed : name /AllEvents
  /Interests : array [2]
      0 : event interest(/Damaged)
          /Action : null null
          /Canvas : canvas can(0,0,1152,900)
          /ClientData : dict <1/10>
          /CallBack : array {6}
              0 : operator 'newprocessgroup'
              1 : operator 'damagepath'
              2 : operator 'clipcanvas'
              3 : name PaintRoot
              4 : operator 'newpath'
              5 : operator 'clipcanvas'
          /Exclusivity : boolean false
          /Interest : null null
          /IsInterest : boolean true
          /IsQueued : boolean false
          /KeyState : array [0]
          /Name : name /Damaged
          /Priority : integer 0
          /Process : process proc(input_wait,'awaitevent')
          /Serial : real 0.1621
          /TimeStamp : integer 0
          /Timestamp : real 0.275
          /XLocation : integer -183
          /YLocation : integer 172
      1 : event interest(/RightMouseButton)
          /Action : name /DownTransition
          /Canvas : canvas can(0,0,1152,900)
          /ClientData : dict <1/10>
          /CallBack : array {3}
              0 : array {4}
                  0 : operator 'newprocessgroup'
                  1 : name /showat
                  2 : name rootmenu
                  3 : operator 'send'
              1 : operator 'fork'
              2 : operator 'pop'
          /Exclusivity : boolean false
          /Interest : null null
          /IsInterest : boolean true
          /IsQueued : boolean false
          /KeyState : array [0]
          /Name : name /RightMouseButton
          /Priority : integer 0
          /Process : process proc(input_wait,'awaitevent')
          /Serial : real 0.071
          /TimeStamp : integer 0
          /Timestamp : real 0.275
          /XLocation : integer -183
          /YLocation : integer 172
  /Mapped : boolean true
  /Parent : null null
  /Retained : boolean false
  /SaveBehind : boolean false
  /TopCanvas : canvas can(84,682,207,207)
  /TopChild : canvas can(84,682,207,207)
  /Transparent : boolean false
```



```
process •
proc(input_wait,'awaitevent')
  /DictionaryStack : array [8]
      0 : dict <910/2000>
      1 : dict <100/200>
      2 : Object <10/200>
      3 : Item <41/200>
      4 : LabeledItem <16/200>
      5 : StructItem <194/200>
      6 : .StructItem <69/200>
  /ErrorCode : name /none
  /ErrorDetailLevel : integer 1
  /Execee : operator 'awaitevent'
  /ExecutionStack : array [10]
      0 : array {66}
      1 : integer 64
      2 : array {10}
      3 : integer 4
      4 : array {1}
          0 : name eventloop
      5 : integer 1
      6 : array {2}
          0 : array {1}
          1 : operator 'loop'
      7 : integer 2
      8 : array {1}
          0 : operator 'awaitevent'
      9 : integer 1
  /Interests : array [1]
      0 : event interest(<3/20>)
  /OperandStack : array [2]
      0 : .StructItem <69/200>
      1 : process proc(input_wait,'awaitevent')
  /SendContexts : array [0]
  /State : name /input_wait
```

Figure 5: View Characteristics

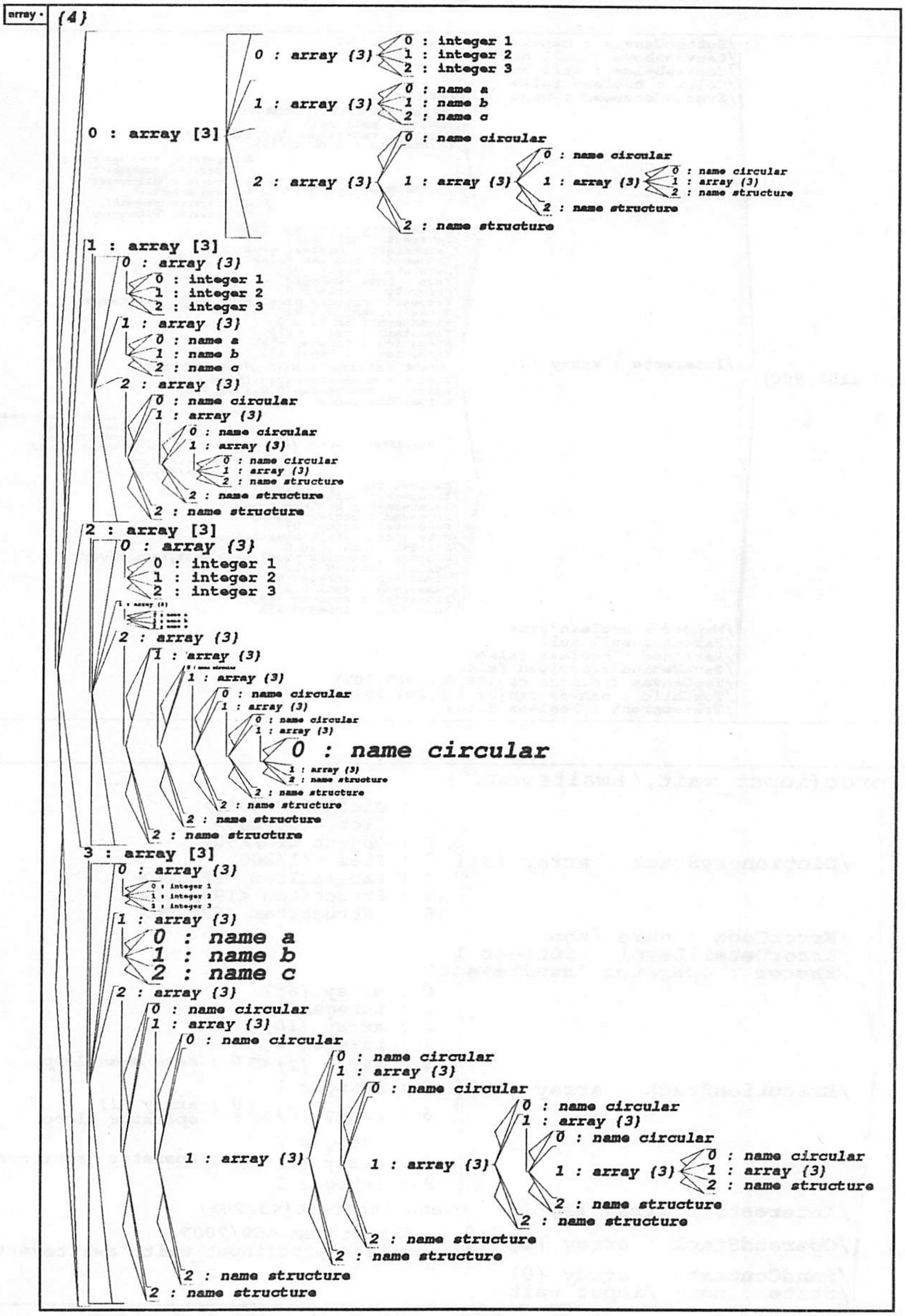
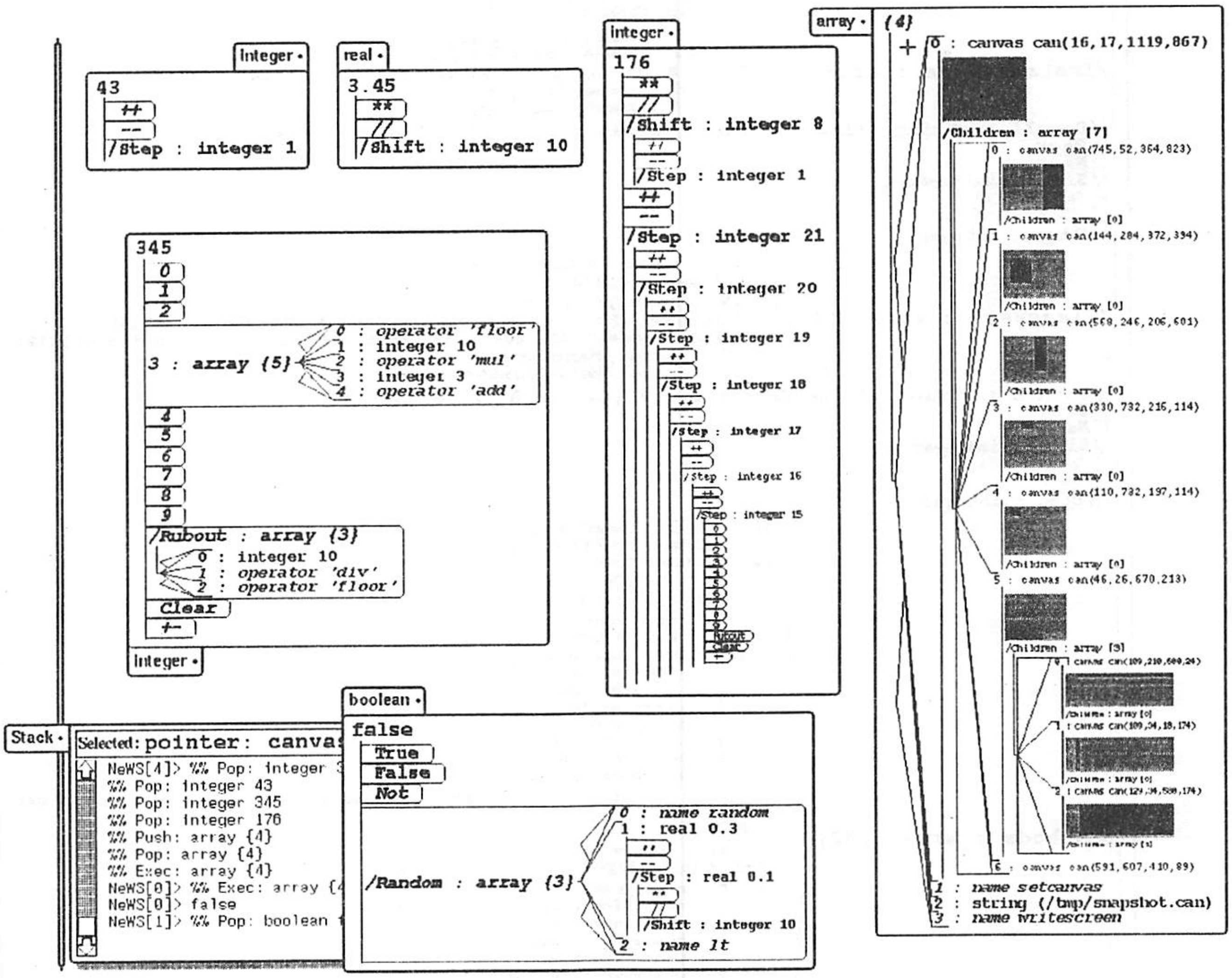


Figure 6: Peripheral Controlers



```
43
/Step : integer 1
```

```
3.45
/Shift : integer 10
```

```
345
3 : array {5}
    0 : operator 'floor'
    1 : integer 10
    2 : operator 'mul'
    3 : integer 3
    4 : operator 'add'
/Rubout : array {3}
    0 : integer 10
    1 : operator 'div'
    2 : operator 'floor'
```

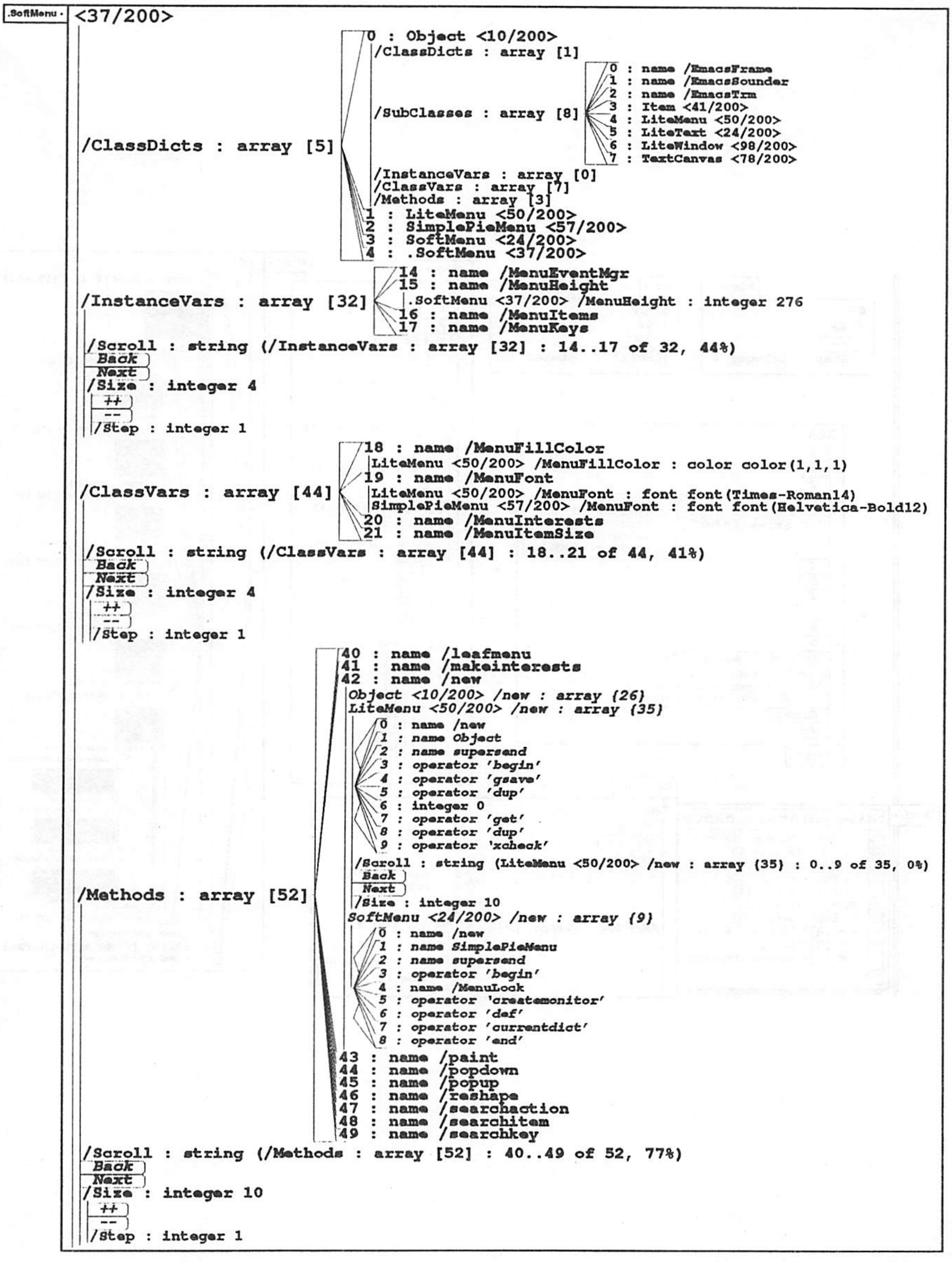
```
176
/Shift : integer 8
/Step : integer 1
/Step : integer 21
/Step : integer 20
/Step : integer 19
/Step : integer 18
/Step : integer 17
/Step : integer 16
/Step : integer 15
Rubout
Clear
```

```
{4}
+ 0 : canvas can(16,17,1119,867)
    /Children : array [7]
        0 : canvas can(745,52,364,823)
            /Children : array [0]
        1 : canvas can(144,284,372,394)
            /Children : array [0]
        2 : canvas can(568,246,206,601)
            /Children : array [0]
        3 : canvas can(330,732,216,114)
            /Children : array [0]
        4 : canvas can(110,732,197,114)
            /Children : array [0]
        5 : canvas can(46,26,670,213)
            /Children : array [3]
                0 : canvas can(109,210,600,24)
                    /Children : array [0]
                1 : canvas can(109,34,18,174)
                    /Children : array [0]
                2 : canvas can(129,34,588,174)
                    /Children : array [1]
        6 : canvas can(591,607,410,89)
  1 : name setcanvas
  2 : string (/tmp/snapshot.can)
  3 : name writescreen
```

```
Selected: pointer: canvas
NeWS[4]> %% Pop: integer 3
%% Pop: integer 43
%% Pop: integer 345
%% Pop: integer 176
%% Push: array {4}
%% Pop: array {4}
%% Exec: array {4}
NeWS[0]> %% Exec: array {4
NeWS[0]> false
NeWS[1]> %% Pop: boolean f
```

```
false
/Random : array {3}
    0 : name random
    1 : real 0.3
        /Step : real 0.1
        /Shift : integer 10
    2 : name lt
```

Figure 7: Class Editor: rootmenu



```
.SoftMenu · <37/200>
                        0 : Object <10/200>
                         /ClassDicts : array [1]
                                                   0 : name /EmacsFrame
                                                   1 : name /EmacsSounder
                                                   2 : name /EmacsTrm
                         /SubClasses : array [8]   3 : Item <41/200>
                                                   4 : LiteMenu <50/200>
                                                   5 : LiteText <24/200>
/ClassDicts : array [5]                            6 : LiteWindow <98/200>
                                                   7 : TextCanvas <78/200>
                         /InstanceVars : array [0]
                         /ClassVars : array [7]
                         /Methods : array [3]
                        1 : LiteMenu <50/200>
                        2 : SimplePieMenu <57/200>
                        3 : SoftMenu <24/200>
                        4 : .SoftMenu <37/200>
                          14 : name /MenuEventMgr
                          15 : name /MenuHeight
/InstanceVars : array [32]  .SoftMenu <37/200> /MenuHeight : integer 276
                          16 : name /MenuItems
                          17 : name /MenuKeys
 /Scroll : string (/InstanceVars : array [32] : 14..17 of 32, 44%)
 Back
 Next
 /Size : integer 4
  ++
  --
  /Step : integer 1
                        18 : name /MenuFillColor
                         LiteMenu <50/200> /MenuFillColor : color color(1,1,1)
                        19 : name /MenuFont
/ClassVars : array [44]  LiteMenu <50/200> /MenuFont : font font(Times-Roman14)
                         SimplePieMenu <57/200> /MenuFont : font font(Helvetica-Bold12)
                        20 : name /MenuInterests
                        21 : name /MenuItemSize
 /Scroll : string (/ClassVars : array [44] : 18..21 of 44, 41%)
 Back
 Next
 /Size : integer 4
  ++
  --
  /Step : integer 1
                       40 : name /leafmenu
                       41 : name /makeinterests
                       42 : name /new
                        Object <10/200> /new : array {26}
                        LiteMenu <50/200> /new : array {35}
                             0 : name /new
                             1 : name Object
                             2 : name supersend
                             3 : operator 'begin'
                             4 : operator 'gsave'
                             5 : operator 'dup'
                             6 : integer 0
                             7 : operator 'get'
                             8 : operator 'dup'
                             9 : operator 'xcheck'
                         /Scroll : string (LiteMenu <50/200> /new : array {35} : 0..9 of 35, 0%)
                         Back
                         Next
/Methods : array [52]    /Size : integer 10
                        SoftMenu <24/200> /new : array {9}
                             0 : name /new
                             1 : name SimplePieMenu
                             2 : name supersend
                             3 : operator 'begin'
                             4 : name /MenuLock
                             5 : operator 'createmonitor'
                             6 : operator 'def'
                             7 : operator 'currentdict'
                             8 : operator 'end'
                       43 : name /paint
                       44 : name /popdown
                       45 : name /popup
                       46 : name /reshape
                       47 : name /searchaction
                       48 : name /searchitem
                       49 : name /searchkey
 /Scroll : string (/Methods : array [52] : 40..49 of 52, 77%)
 Back
 Next
 /Size : integer 10
  ++
  --
  /Step : integer 1
```

Figure 8: Pseudo Scientific Visualizer: rootmenu

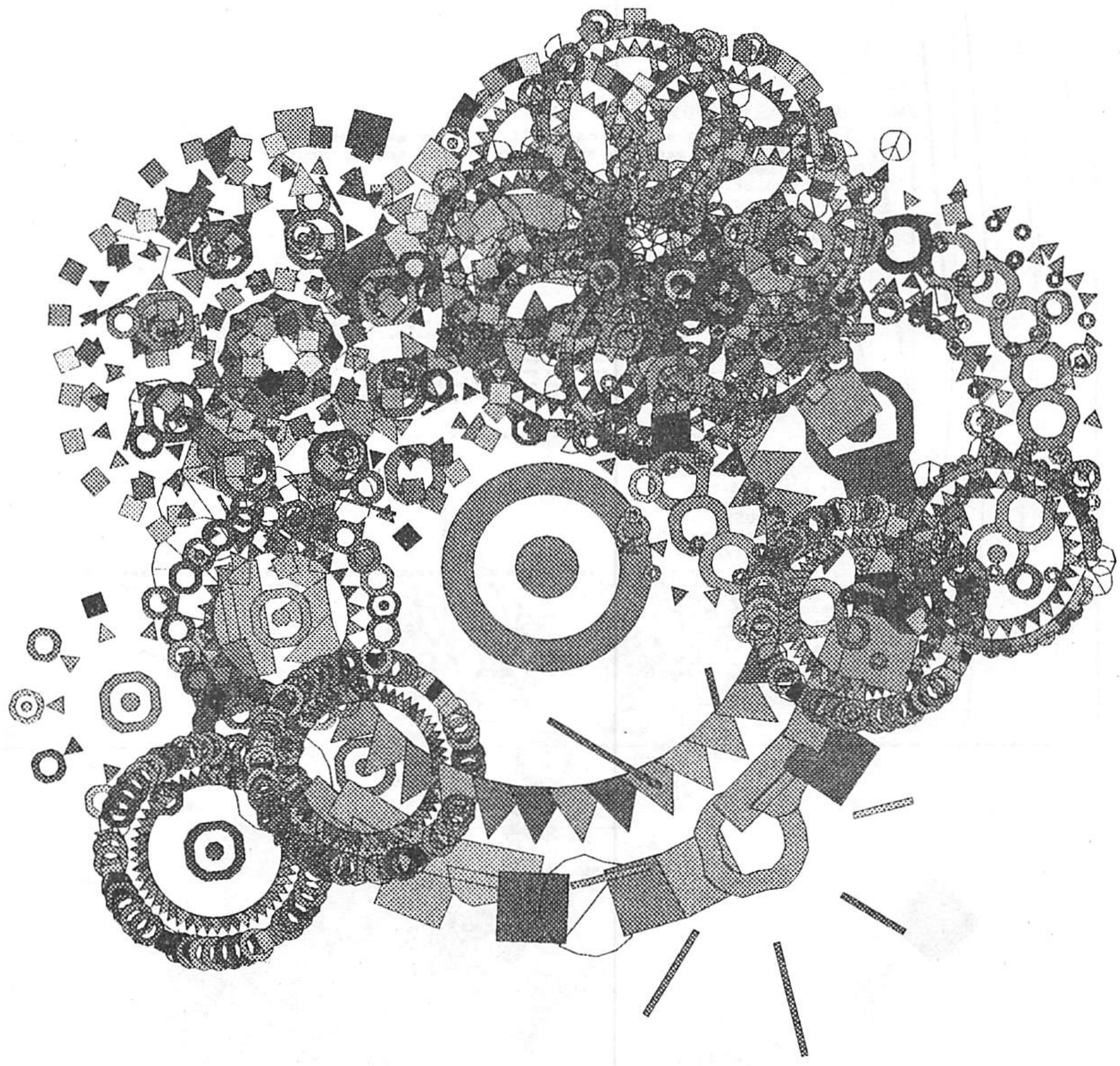
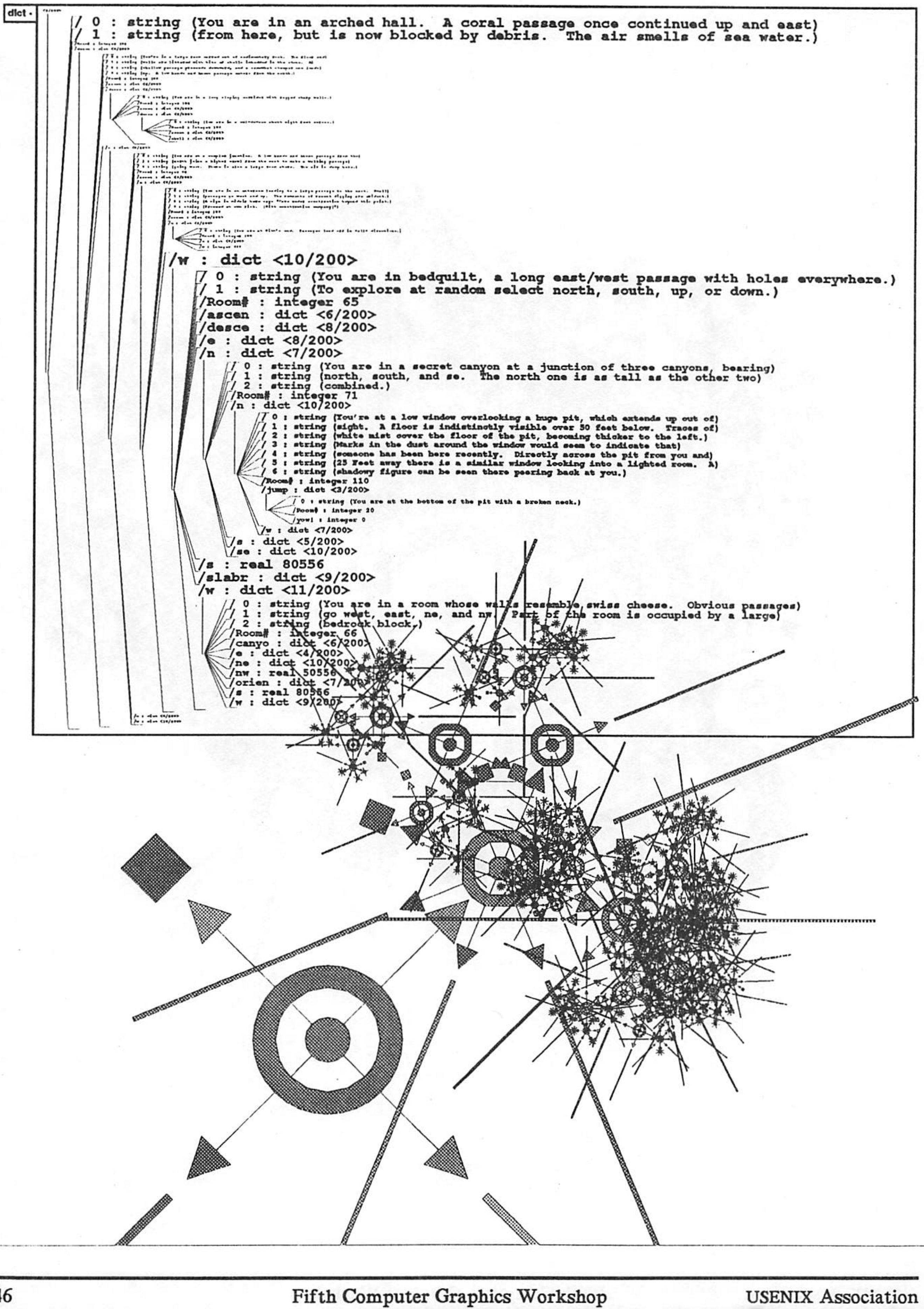
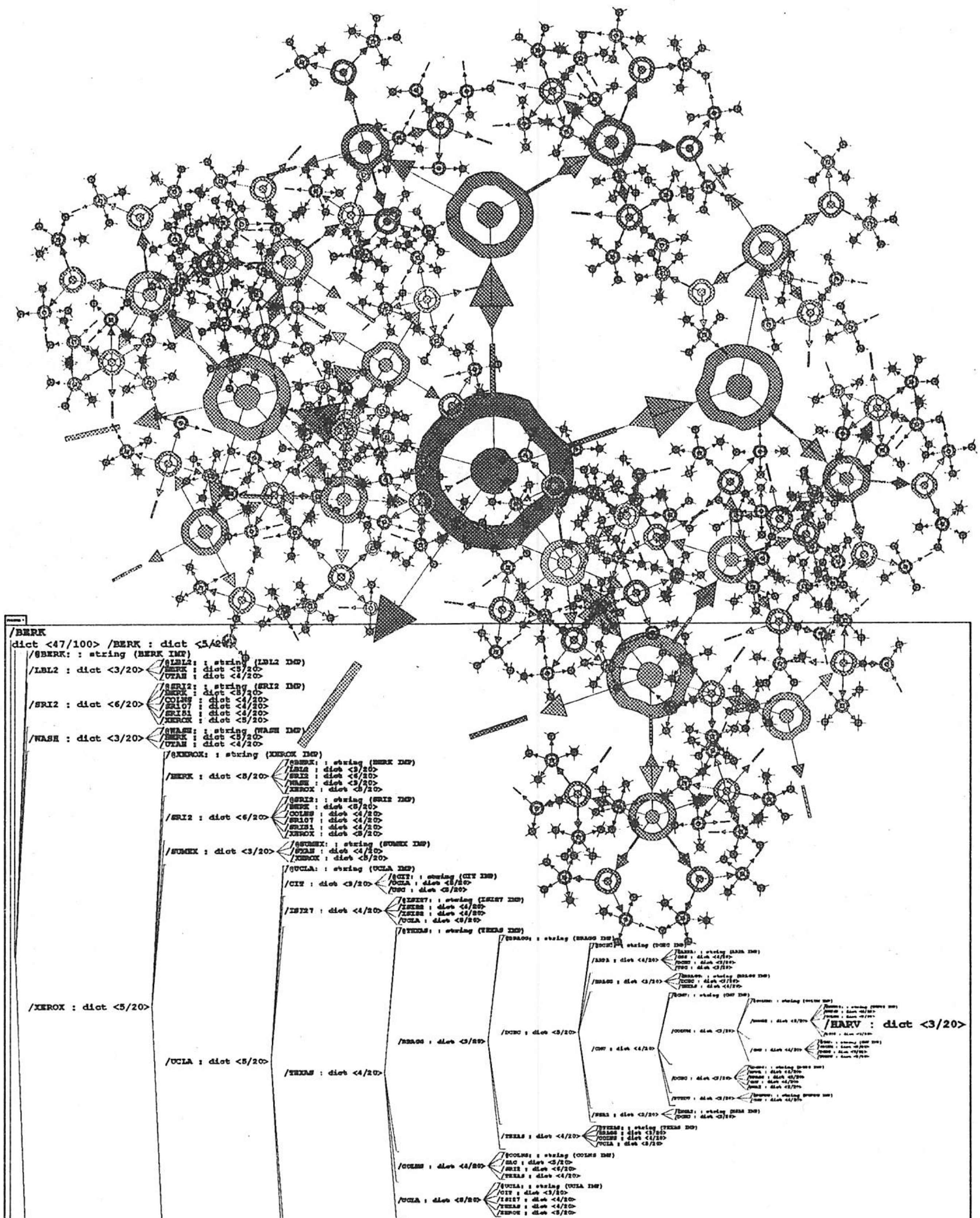


Figure 9: Map of Adventure



```
/ 0 : string (You are in an arched hall.  A coral passage once continued up and east)
/ 1 : string (from here, but is now blocked by debris.  The air smells of sea water.)
/w : dict <10/200>
/ 0 : string (You are in bedquilt, a long east/west passage with holes everywhere.)
/ 1 : string (To explore at random select north, south, up, or down.)
/Room# : integer 65
/ascen : dict <6/200>
/desce : dict <8/200>
/e : dict <8/200>
/n : dict <7/200>
/ 0 : string (You are in a secret canyon at a junction of three canyons, bearing)
/ 1 : string (north, south, and se.  The north one is as tall as the other two)
/ 2 : string (combined.)
/Room# : integer 71
/n : dict <10/200>
/ 0 : string (You're at a low window overlooking a huge pit, which extends up out of)
/ 1 : string (sight.  A floor is indistinctly visible over 50 feet below.  Traces of)
/ 2 : string (white mist cover the floor of the pit, becoming thicker to the left.)
/ 3 : string (Marks in the dust around the window would seem to indicate that)
/ 4 : string (someone has been here recently.  Directly across the pit from you and)
/ 5 : string (25 Feet away there is a similar window looking into a lighted room.  A)
/ 6 : string (shadowy figure can be seen there peering back at you.)
/Room# : integer 110
/jump : dict <3/200>
/ 0 : string (You are at the bottom of the pit with a broken neck.)
/Room# : integer 20
/yow! : integer 0
/w : dict <7/200>
/s : dict <5/200>
/se : dict <10/200>
/s : real 80556
/slabr : dict <9/200>
/w : dict <11/200>
/ 0 : string (You are in a room whose walls resemble swiss cheese.  Obvious passages)
/ 1 : string (go west, east, ne, and nw.  Part of the room is occupied by a large)
/ 2 : string (bedrock block.)
/Room# : integer 66
/canyo : dict <6/200>
/e : dict <4/200>
/ne : dict <10/200>
/nw : real 50556
/orien : dict <7/200>
/s : real 80556
/w : dict <9/200>
```

Figure 10: ARPANET, around Berkeley IMP



```
/BERK
dict <47/100> /BERK : dict <5/20>
    /@BERK: : string (BERK IMP)
    /LBL2 : dict <3/20>   /@LBL2: : string (LBL2 IMP)
                          /BERK : dict <5/20>
                          /UTAH : dict <4/20>
    /SRI2 : dict <6/20>   /@SRI2: : string (SRI2 IMP)
                          /BERK : dict <5/20>
                          /COLNS : dict <4/20>
                          /SR107 : dict <4/20>
                          /SRI51 : dict <4/20>
                          /XEROX : dict <5/20>
    /WASH : dict <3/20>   /@WASH: : string (WASH IMP)
                          /BERK : dict <5/20>
                          /UTAH : dict <4/20>
    /XEROX : dict <5/20>
        /@XEROX: : string (XEROX IMP)
        /BERK : dict <5/20>    /@BERK: : string (BERK IMP)
                               /LBL2 : dict <3/20>
                               /SRI2 : dict <6/20>
                               /WASH : dict <3/20>
                               /XEROX : dict <5/20>
        /SRI2 : dict <6/20>    /@SRI2: : string (SRI2 IMP)
                               /BERK : dict <5/20>
                               /COLNS : dict <4/20>
                               /SR107 : dict <4/20>
                               /SRI51 : dict <4/20>
                               /XEROX : dict <5/20>
        /SUMEX : dict <3/20>   /@SUMEX: : string (SUMEX IMP)
                               /STAN : dict <4/20>
                               /XEROX : dict <5/20>
        /UCLA : dict <5/20>
            /@UCLA: : string (UCLA IMP)
            /CIT : dict <3/20>     /@CIT: : string (CIT IMP)
                                   /UCLA : dict <5/20>
                                   /USC : dict <3/20>
            /ISI27 : dict <4/20>   /@ISI27: : string (ISI27 IMP)
                                   /ISI22 : dict <4/20>
                                   /ISI52 : dict <4/20>
                                   /UCLA : dict <5/20>
            /TEXAS : dict <4/20>
                /@TEXAS: : string (TEXAS IMP)
                /BRAGG : dict <3/20>
                    /@BRAGG: : string (BRAGG IMP)
                    /DCEC : dict <3/20>
                        /@DCEC: : string (DCEC IMP)
                        /ARPA : dict <4/20>    /@ARPA: : string (ARPA IMP)
                                               /GSS : dict <4/20>
                                               /DCEC : dict <3/20>
                                               /USC : dict <3/20>
                        /BRAGG : dict <3/20>   /@BRAGG: : string (BRAGG IMP)
                                               /DCEC : dict <3/20>
                                               /TEXAS : dict <4/20>
                        /CMU : dict <4/20>
                            /@CMU: : string (CMU IMP)
                            /[illegible] : dict <3/20>
                                /[illegible] : string ([illegible] IMP)
                                /[illegible] : dict <3/20>   [illegible]
                                                             /HARV : dict <3/20>
                                                             [illegible]
                                /[illegible] : dict <4/20>   [illegible]
                            /DCEC : dict <3/20>   [illegible]
                            /[illegible] : dict <3/20>   [illegible]
                        /[illegible] : dict <2/20>   /@[illegible]: : string ([illegible] IMP)
                                                     /DCEC : dict <3/20>
                    /TEXAS : dict <4/20>   /@TEXAS: : string (TEXAS IMP)
                                           /BRAGG : dict <3/20>
                                           /COLNS : dict <4/20>
                                           /UCLA : dict <5/20>
                /COLNS : dict <4/20>   /@COLNS: : string (COLNS IMP)
                                       /SAC : dict <3/20>
                                       /SRI2 : dict <6/20>
                                       /TEXAS : dict <4/20>
                /UCLA : dict <5/20>    /@UCLA: : string (UCLA IMP)
                                       /CIT : dict <3/20>
                                       /ISI27 : dict <4/20>
                                       /TEXAS : dict <4/20>
                                       /XEROX : dict <5/20>
        /XEROX : dict <5/20>   /@XEROX: : string (XEROX IMP)
                               /BERK : dict <5/20>
                               /SRI2 : dict <6/20>
                               /SUMEX : dict <3/20>
                               /UCLA : dict <5/20>
```

The USENIX Association

The USENIX Association is a not-for-profit organization of those interested in UNIX and UNIX-like systems. It is dedicated to fostering and communicating the development of research and technological information and ideas pertaining to advanced computing systems, to the monitoring and encouragement of continuing innovation in advanced computing environments, and to the provision of a forum where technical issues are aired and critical thought exercised so that its members can remain current and vital.

To these ends, the Association conducts large semi-annual technical conferences and sponsors workshops concerned with varied special-interest topics; publishes proceedings of those meetings; publishes a bimonthly newsletter *;login:*; produces a quarterly technical journal, *Computing Systems*; serves as coordinator of an exchange of software; and distributes 4.3BSD manuals and 2.10BSD tapes. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts. Most recently, the Association created UUNET Communications Services, Inc., a separate not-for-profit organization offering electronic communications services to those wishing to participate in the UNIX milieu.

Computing Systems, published quarterly in conjunction with the University of California Press, is a refereed scholarly journal devoted to chronicling the development of advanced computing systems. It uses an aggressive review cycle providing authors with the opportunity to publish new results quickly, usually within six months of submission.

The USENIX Association intends to continue these and other projects, and will focus new energies on expanding the Association's activities in the areas of outreach to universities and students, improving the technical community's visibility and stature in the computing world, and continuing to improve its conferences and workshops.

The Association was formed in 1975 and incorporated in 1980 to meet the needs of the UNIX technical community. It is governed by a Board of Directors elected biennially.

There are four classes of membership in the Association, differentiated primarily by the fees paid and services provided.

For further information about membership or to order publications, contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

Telephone: 415 528-8649
Email: office@usenix.org